Cross layer reliability estimation for digital systems

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer Engineering (29^{th}cycle)

Cross layer reliability estimation for digital systems

By

Alessandro Vallero

******

Supervisor(s):
Prof. Stefano Di Carlo, Supervisor

Doctoral Examination Committee:
Prof. Regis Leveugle , Referee, TIMA Grenoble
Prof. Marco Ottavi, Referee, University of "Rome Tor Vergata"
Prof. Ramon Canal, UPC Barcelona
Prof. Lorena Angel, TIMA Grenoble
Prof. Dimitris Gizopoulos, University of Athens

Politecnico di Torino
2017

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Alessandro Vallero
2017

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my grandparents who always open my mind toward new horizons

Acknowledgements

I would like to acknowledge Prof. Stefano Di Carlo, at first. He made my Ph.D. a pleasant journey during which I had the chance to learn so many things from him, from both a professional and a human point of view. Secondly, I would like to thank my mother, Linda, and my sister, Greta. They supported me emotionally, even in very stressful moments. I acknowledge Computer Architecture Lab of University of Athens. It was a pleasure to work together. During my visiting you made me feel like at home. Thank you Prof. Dimitris Gizopoulos, Sotiris Tselonis, Manolis Kaliorakis, Athanasios Chatzdimitriou, Giorgos Papadimitriou. In addition, I would like to thank all the CLERECO partners with which I collaborated for my Ph.D. activities. I would also like to acknowledge my housemates, Ivan and Daniele, who suffered me for more than two long years! Finally, I thank all the guys of my lab at Politecnico di Torino (Lab 6) who always laughed at my jokes and contributed to create a pleasant atmosphere.

Abstract

Forthcoming manufacturing technologies hold the promise to increase multifuctional computing systems performance and functionality thanks to a remarkable growth of the device integration density. Despite the benefits introduced by this technology improvements, reliability is becoming a key challenge for the semiconductor industry. With transistor size reaching the atomic dimensions, vulnerability to unavoidable fluctuations in the manufacturing process and environmental stress rise dramatically. Failing to meet a reliability requirement may add excessive re-design cost to recover and may have severe consequences on the success of a product.

One of the open challenges for future technologies is building "dependable" systems on top of unreliable components, which will degrade and even fail during normal lifetime of the chip. Conventional design techniques are highly inefficient. They expend significant amount of energy to tolerate the device unpredictability by adding safety margins to a circuit's operating voltage, clock frequency or charge stored per bit. Unfortunately, the additional cost introduced to compensate unreliability are rapidly becoming unacceptable in today's environment where power consumption is often the limiting factor for integrated circuit performance, and energy efficiency is a top concern.

Attention should be payed to tailor techniques to improve the reliability of a system on the basis of its requirements, ending up with cost-effective solutions favoring the success of the product on the market. Cross-layer reliability is one of the most promising approaches to achieve this goal. Cross-layer reliability techniques take into account the interactions between the layers composing a complex system (i.e., technology, hardware and software layers) to implement efficient cross-layer fault mitigation mechanisms. Fault tolerance mechanism are carefully implemented at different layers starting from the technology up to the software layer to carefully

optimize the system by exploiting the inner capability of each layer to mask lower level faults.

For this purpose, cross-layer reliability design techniques need to be complemented with cross-layer reliability evaluation tools, able to precisely assess the reliability level of a selected design early in the design cycle. Accurate and early reliability estimates would enable the exploration of the system design space and the optimization of multiple constraints such as performance, power consumption, cost and reliability.

This Ph.D. thesis is devoted to the development of new methodologies and tools to evaluate and optimize the reliability of complex digital systems during the early design stages. More specifically, techniques addressing hardware accelerators (i.e., FPGAs and GPUs), microprocessors and full systems are discussed. All developed methodologies are presented in conjunction with their application to real-world use cases belonging to different computational domains.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms / Abbreviations

ACE Architectural Correct Execution

AFTS Autonomous Fault-Tolerant Systems

ASIC Application Specific Integrated Circuits

AVF Architectural Vulnerability Factor

BN Bayesian Network

BRA Bayesian Reliability Analysis

CL Component Library

CPT Conditional Probability Table

CPU Central Processing Unit

CU Compute Unit

DPR Dynamic Partial Reconfiguration

DUE Detectable Unrecoverable Error

EO Extremal Optimization

EPF Execution Per Failure

ES Embedded Systems

FI Fault Injection

FPGA Field Programmable Gate Array

GPGPU General Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

HPC High Performance Computing

HwD Hardware Domain

IPF Instructions Per Failure

IPH Injections Per Hour

IRE Injection Runs up to the End

IRS Injection Runs up to the Software level

ISA Instruction Set Architecture

MBU Multiple Bit Upset

MLRA Microarchitectural Level Reliability Analysis

OS Operating System

PDF Probability Density Function

RTL Register Transfer Level

RTRA Register Transfer Reliability Analysis

RTRA − *COP* RTRA CPU Observation Points

RTRA − *COP* RTRA Software Observation Points

SD System Domain

SDC Silent Data Corruption

SER Soft Error Rate

SEU Single Event Upset

SFB Software Faulty Behavior

SFM Software Fault Model

SIMD Single Instruction Multiple Data

SM Software Module

SUE System Under Evaluation

SwD Software Domain

TD Technology Domain

TMR Triple Modular Redundancy

TTM Time To Market

About the author

Alessandro Vallero received his Bachelor of Science and Master of Science degree in Electronics Engineering from Politecnico di Torino (Italy), respectively, in 2008 and 2013. He also received his Master of Science in Electrical and Computer Engineering from University of Illinois at Chicago (US) in 2014.

Alessandro started his PhD in January 2013. His major interests are reliability analysis of digital systems and reconfigurable computing. During the PhD activity he contributed actively to the CLERECO FP7 European project. Moreover he received a HiPEAC Collaboration grants to visit University of Athens (Greece) and collaborate to the development of reliability analaysis techniques for general purpose computing on GPUs.

List of publications

This is the list of the papers that Alessandro published during its Ph.D.:

- Di Carlo S.; Gambardella G.; Prinetto P.; Rolfo D.; Trotta P.; Vallero A. “A novel methodology to increase fault tolerance in autonomous FPGA-based systems”, IEEE 20th International On-Line Testing Symposium (IOLTS), Platja d’Aro, Girona (ES), 7-9 July 2014. pp. 87-92.

- Di Carlo, S.; Vallero, A.; Gizopoulos, D.; Di Natale, G.; Gonzalez, A.; Canal, R.; Mariani, R.; Pipponzi, M.; Grasset, A.; Bonnot, F.; Reichenbach, F.; Rafiq, G.; Loekstad T. "Cross-layer early reliability evaluation: Challenges and promise", IEEE 20th International On-Line Testing Symposium (IOLTS), Platja d'Aro, Girona (ES), 7-9 July 2014. pp. 228-233.

- Di Carlo, S.; Vallero, A.; Gizopoulos, D.; Di Natale, G.; Grasset, A.; Mariani, R.; Reichenbach, F. "Cross-Layer Early Reliability Evaluation for the Computing cOntinuum", 17th Euromicro Conference on Digital System Design (DSD), Verona, IT, 27-29 Aug. 2014. pp. 199-205.

- Alessandro VALLERO, Alessandro SAVINO, Gianfranco POLITANO, Stefano DI CARLO (Politecnico di Torino - Italy), Sotiris TSELONIS, Nikos FOUTRIS, Manolis KALIORAKIS, Dimitris GIZOPOULOS (University of Athens - Greece), "A bayesian model for system level reliability estimation", 20th IEEE European Test Symposium.

- A.Vallero, A.Savino (Politecnico di Torino), S.Tselonis, N.Foutris, M.Kaliorakis (University of Athens), G.Politano (Politecnico di Torino), D.Gizopoulos (University of Athens), S.Di Carlo (Politecnico di Torino), "Bayesian Network Early Reliability Evaluation Analysis for both Permanent and Transient Faults", 21st IEEE International On-Line Testing Symposium.

- A.Vallero, S.Tselonis, N.Foutris, M.Kaliorakis, M. Kooli, A.Savino, G.Politano, A.Bosio, G. Di Natale, D.Gizopoulos, S.Di Carlo, "Cross-Layer Reliability Evaluation, moving from the Hardware Architecture to the System level: a CLERECO EU Project overview", Microprocessors and Microsystems, 2015.

- Vallero, A.; Savino, A.; Politano, G.; Di Carlo, S.; Chatzidimitriou, A.; Tselonis, S.; Kaliorakis, M.; Gizopoulos, D.; Riera, M.; Canal, R.; Gonzalez, A.; Kooli, M.; Bosio, A.; Di Natale, G. "Early Component-Based System Reliability Analysis for Approximate Computing Systems" 2nd Workshop On Approximate Computing (WAPCO), Prague, CZ, 20 Jan. 2016.

- A. Savino; S. Di Carlo; A. Vallero; G. Politano; D. Gizopoulos; A. Evans, "RIIF-2: Toward the next generation reliability information interchange format", 2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS).

- A. Vallero, A. Savino, G. Politano, S. Di Carlo, A. Chatzidimitriou, S. Tselonis, M. Kaliorakis, D. Gizopoulos, M. Riera, R. Canal, A. Gonzalez, M. Kooli, A. Bosio, G. Di Natale, "Cross-Layer System Reliability Assessment Framework for Hardware Faults" IEEE International Test Conference (ITC), 2016.

- Alessandro Vallero, Sotiris Tselonis, Dimitris Gizopoulos, and Stefano Di Carlo, "Microarchitecture Level Reliability Comparison of Modern GPU Designs: first findings" Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on.

- Alessandro Vallero, Dimitris Gizopoulos, and Stefano Di Carlo, "SIFI: AMD Southern Islands GPU Microarchitectural Level Fault Injector" 2017 IEEE 23nd International Symposium on On-Line Testing and Robust System Design (IOLTS) IEEE, 2017.

Chapter 1

Introduction

1.1 Motivation

Information technology is at the core of our society and it relies completely on the design of electronic information processing systems. Today's computing is a true continuum that ranges from smartphones to mission-critical datacenter machines, and from desktops to automobiles. On aggregate, these computing devices represent a total addressable market approaching a billion processors a year, which is expected to explode to more than two billion per year before 2020.

The computing industry move towards the *computing continuum* means that the same key technologies and industrial players will act across all computing segments: airplanes, automobile, buildings, health instruments, smartphones, tablets, desktops, servers, datacenters, clouds, high-performance computing (HPC), etc. Therefore, in the near future we will see embedded systems (ES) with HPC performance and functionalities, HPC systems used in time and safety critical applications, cloud resources used with very different business models, etc.

For more than three decades, industry has evolved by roughly doubling the device density (and corresponding performance) every two years following Moore's law. However, future device integration technology is expected to dramatically reduce the device quality, and therefore the operational reliability of circuits: as the transistors and wires shrink, they show both larger differences in behavior although they are designed to be identical (device variability, manufacturing defects, aging), and higher susceptibility to transient and permanent faults (soft errors, wear-out). The

great challenge for future technologies is building "dependable" systems on top of unreliable components, which will degrade and even fail during normal lifetime of the chip. Dependability is however not a quantitative term which is instead quantified by *reliability* (i.e., continuity of correct service), *availability* (i.e., readiness for correct service), *safety* (absence of catastrophic consequences on the user(s) and the environment), *maintainability* (i.e., ability to undergo modifications and repair). Reliability is the most often used concept in the industry and academia. Therefore, without loss of generality, the word reliability will be employed for the rest of the thesis when referring to all concepts related to dependability.

First studies addressing reliability with an engineering approach date back to World War II and they were mainly devoted towards understanding the behavior of electronic devices in presence of radiation. Alongside the military field, reliability for digital circuits found application in the space domain as well as in medical fields where radiation are involved. In recent years, as technology has progressed, new reliability issues started to manifest in consumer devices. In fact, from one side there was a reduction of cost associated to the production of digital devices benefiting of the shrinking of circuits thanks to technological improvements; from the other side reliability problems arose due to the manufacturing process and the dramatic shrinking of the feature sizes.

In the industrial domain, nowadays, reliability is a fundamental aspect when a new system is designed, as it influence its cost and success on the market. Some techniques can be adopted to improve reliability of a digital circuit, but they come at a price: they increase manufacturing and engineering cost. Moreover they usually have a negative impact on performance, power consumption and area of the designed products. Conventional design techniques expend significant amount of energy to tolerate the device unpredictability by adding safety margins to a circuit's operating voltage, clock frequency or charge stored per bit. However, the rising energy cost needed to compensate for increasing unpredictability are rapidly becoming unacceptable in today's environment where power consumption is often the limiting factor for integrated circuit performance, and energy efficiency is a top concern.
On one hand, if reliability was not taken into account, the designed system would be very cheap despite a short lifetime. As a result, nobody would buy it, or, even worse, the system would not be suitable to be employed as it would not meet the reliability requirements. On the other hand, if reliability led to an over-designed

system, a system with many reliability improvements, the lifetime of this product would be excellent. However, the associated cost would grow exponentially alongside the system performance degradation, making this product not competitive with the others. In some cases, a negative return of investment would be unavoidable because of over-design.

Much attention should be payed to tailor techniques for reliability improvements on the basis of system needs, ending up with a cost-effective system successful on the market. Finding the right balance among system reliability, cost and performance is not an easy task and it usually takes some time during the different stages of the design, thus increasing the time-to-market of the product and the possibility to have a negative return of investment.

Implementing systems belonging to the computing continuum in this era, where low reliability threatens to end the benefits of feature size reduction, requires a holistic approach across different computing disciplines, across computing system layers and across computing market segments to have a unique reliability assessment methodology. The main goal of this thesis is to propose a framework addressing the problem of having an early, fast, and accurate evaluation of computing systems reliability to support design decisions for hardware and software reliability enhancing mechanisms in the system. Such a framework will contribute to the continuation of technology scaling benefits harnessing for several decades. Moreover, it will also enable the implementation of the computing continuum that societal services demand.

Cross-layer reliability is one of the most promising approach to reliability, leveraging the many commonalities between different application domains and computing disciplines in the next generation computing continuum. This methodology aims at providing a reliability analysis despite differences among application domains and computing systems by identifying their common aspects. It takes into account the abstraction layers a system is composed of (i.e., technology, hardware and software layers) and addresses their interaction. Consequently, resorting to a cross-layer approach for reliability estimation enables the exploration and optimization of the design space (performance, portability, energy efficiency, dependability, real-time responsiveness) across different domains.

The benefits of an early and accurate methodology to estimate the reliability of a system are many. First, a tool that yields early estimate allows designers to reduce the

time-to-market (TTM) of all products, enabling the fast turnaround of proliferation of computing systems tailored to customers' needs. Nowadays, sometimes the system must be re-designed several times in order to satisfy the target constraints because of a worst-case approach, posing a serious threat to the success of the product. Second, accurate reliability estimates allow developing the required cost- and energy-efficient reliability solutions at the hardware and software layers. Such solutions can be only developed if the expected reliability of the system can be quickly and accurately assessed: (a) at different stages of the design flow (from design concept and early design stages through first silicon validation and eventually during operation in the field), (b) considering the impact of all hardware and software components, and the different modes of operation (use cases) of the system. In detail, cross-layer reliability allows to identify the most efficient protection mechanisms at different system layers (technology, hardware architecture, software), thus exploiting the inner capability of each layer to mask lower level faults. For this purpose, cross-layer reliability design techniques need to be complemented with cross-layer reliability evaluation tools, able to precisely assess the reliability level of a selected design early in the design cycle. Accurate and early reliability estimates enable the exploration of the system design space and the optimization of multiple constraints such as performance, power consumption, cost and reliability. Finally, the development of a reliable and dependable product that employs mechanisms to detect and handle possible faults can reduce the overall life cycle cost.

Finally, targeting the entire emerging computing continuum is an ambitious challenge since it requires methodologies and models to analyze reliability issues belonging to different application fields and considering different components of systems with different complexity. Technology, hardware components (e.g., CPUs, GPUs, DSP, memories, peripherals, accelerators, interconnects), and software (systems and application) from different domains are to be addressed.

1.2 Goals

Traditionally, reliability estimation performed at different stages of the design cycle can lead to "worst-case" decisions and over-designed systems. While the required system reliability can be guaranteed, the cost of the employed reliability mechanisms (in terms or area, energy/power, performance and money) and the design time re-

quired for their integration and evaluation are both excessive. Moreover, standard reliability evaluation approaches strongly rely on massive and time-consuming simulations and/or fault injection campaigns, which are becoming a bottleneck with the increasing complexity of computing systems.

The ultimate goal of my Ph.D. program is the development of new methodologies and tools to evaluate and to optimize the reliability of complex digital systems, in the context of a cross-layer reliability design. In this way, reliability of computing systems can be evaluated at each stage of the design cycle fast and accurately, providing designers with a valuable support for reliability related decision-making process. As a result, improved cost-related characteristics (area, energy/power, and performance) and reduced TTM are also enabled.

The detailed objective of my Ph.D. are:

- To analyze the reliability of the different hardware components at different levels of detail (depending on the design phase), emphasizing the role that the interaction between hardware and software plays in the overall reliability figure of the system.
- To design flexible, fast, and accurate reliability evaluation methodologies and tools for the computing continuum. Offering a more accurate reliability estimation allows to lead system designers to suitable decisions at intermediate design stages, thus effectively reducing the reliability cost and the TTM, consequently impacting the overall life-cycle cost.
- To comprehensively support the reliability decision-making process in computing systems exploiting the proposed reliability evaluation frameworks (methodologies and tools) that leads to meet the reliability constraints of the system avoiding over-design.
- To validate and evaluate in detail the effectiveness of the proposed frameworks in real-world systems belonging to the computing continuum domains.

1.3 Structure of the thesis

An overview on reliability is presented in Chapter 2. In detail, at fist a definition of dependability, faults, errors and failures is given. Then, this chapter introduces the reliability metrics most commonly used when reliability is assessed. Finally, the dependability threats are discussed alongside the related protection mechanisms, focusing mainly on soft errors. This chapter also introduces the key concepts of cross-layer reliability. More specifically, the main advantages enabled by this approach are discussed and analyzed. It follows a deep analysis about the impact on reliability of all the layers a computing system is composed of and their interaction.

My Ph.D. activities covered several aspects of reliability, however they were all developed in the context of cross-layer reliability. In detail, my contribution ranges from reliability studies of hardware accelerators, GPUs, CPUs and digital systems.

Chapter 3 focuses on a methodology to enhance reliability leveraging reconfigurable computing. My contribution was the development of a new methodology to increase fault tolerance of FPGA-based autonomous systems by means of partial reconfiguration. The proposed methodology includes both a new FPGA-based system architecture and a floor planning strategy. Main improvements with respect to state-of-the-art solutions are the reduction (35x) of the required memory to store bitstreams (configurations of the FPGA) and the reduction (4x) of the recovery time, without any loss of performance and tolerating the same number of permanent faults.

A reliability framework for GPUs I developed to assess reliability fast and accurately is introduced in Chapter 4. This framework is based on the publicly available micro-architectural simulator Multi2Sim and it allows to analyze the reliability of applications running on the Southern Islands AMD GPU architecture. The framework allows designers to perform three different kinds of reliability analyses: fault injection, architecturally correct execution analysis and Instruction Correlation analysis. The GPU hardware structures that can be analyzed are: the vector register file, the scalar register file as well as the local data storage. Thanks to its flexibility, this framework can simulate different kinds of AMD GPU chips that differ on their architectural parameters. This enables to easily explore several design spaces and to understand their impact on the reliability, taking into account the executed workload (software application). Moreover, the framework was employed for an

extensive experimental campaign carried out in collaboration with the University of Athens to compare the reliability of AMD GPUs and NVIDIA GPUs. Results of the comparison are also reported and discussed in this chapter.

In Chapter 5, a methodology for cross-layer reliability evaluation of microprocessor based systems affected by soft errors and permanent faults is presented. This methodology takes into account the microprocessor architecture and the executed software along with their interaction at the Instruction Set Architecture level. In detail, it models the sequence of instructions executed by the microprocessor, and the hardware resources used for their execution in the form of a Bayesian Network that can be used to estimate the probability of fault propagation during the execution of the application. This methodology introduces several advantages with respect to current reliability evaluation approaches: a reduction of computation time (some orders of magnitude if compared to micro-architectural fault injection) and high accuracy (it provides results within 3% error margin with respect to results obtained with fault injection).

A methodology to evaluate the reliability of a complex system in the presence of soft errors, allowing the identification of the weakest components of the system is reported in Chapter 6. This approach is based on a Bayesian model of the system enabling a reliability analysis leveraging Bayesian reasoning. The Bayesian model follows the cross-layer paradigm. In details, in the proposed model, network nodes represent the system components, while arcs connecting nodes model relationships between the different system components. This Bayesian Network can be split into three distinct sub-networks, reflecting the main layers of the system: technology, hardware architecture and software. This chapter aims at describing how the model is built and which kind of reliability analysis can be performed resorting to this methodology. In addition, a detailed description of the tools and techniques employed to obtain numbers to populate the model at each system layer is given. Finally, a validation campaign for this methodology is carried out by comparing the obtained results against stat-of-the-art industrial workflows. The analysis concerns reliability assessment of benchmark systems as well as real-life systems belonging to the computing continuum. Chapter 7 introduces a design optimizer based on this Bayesian model. More specifically, the optimization algorithm as well as results are reported.

Finally Chapter 8 concludes the dissertation summarizing the key aspects related to reliability that have emerged and identifies the possible future works in this field.

Chapter 2

Dependability of digital systems

This section overviews basic dependability principles applied to digital systems. At first, the concept of dependability is introduced. The concepts and the taxonomy presented in Section 2.1 serve as the basis for the classification of reliability metrics explained in Section 2.2. Dependability threats and dependability enhancing mechanisms are analyzed in Section 2.3 and Section 2.5.
This thesis focuses mainly on reliability in presence of soft errors. For this reason, specific subsections are added to analyzes into details these aspects.

2.1 Dependability

The term "dependability" covers different aspects of a system, it represents the extent to which a system is expected to operate in compliance to its specifications. Dependability involves several attributes [5]:

- **Availability:** readiness for correct service.
- **Reliability:** continuity of correct service.
- **Safety:** absence of catastrophic consequences on the user(s) and the environment.
- **Integrity:** absence of improper system alterations.
- **Maintainability:** ability to undergo modifications and repairs.

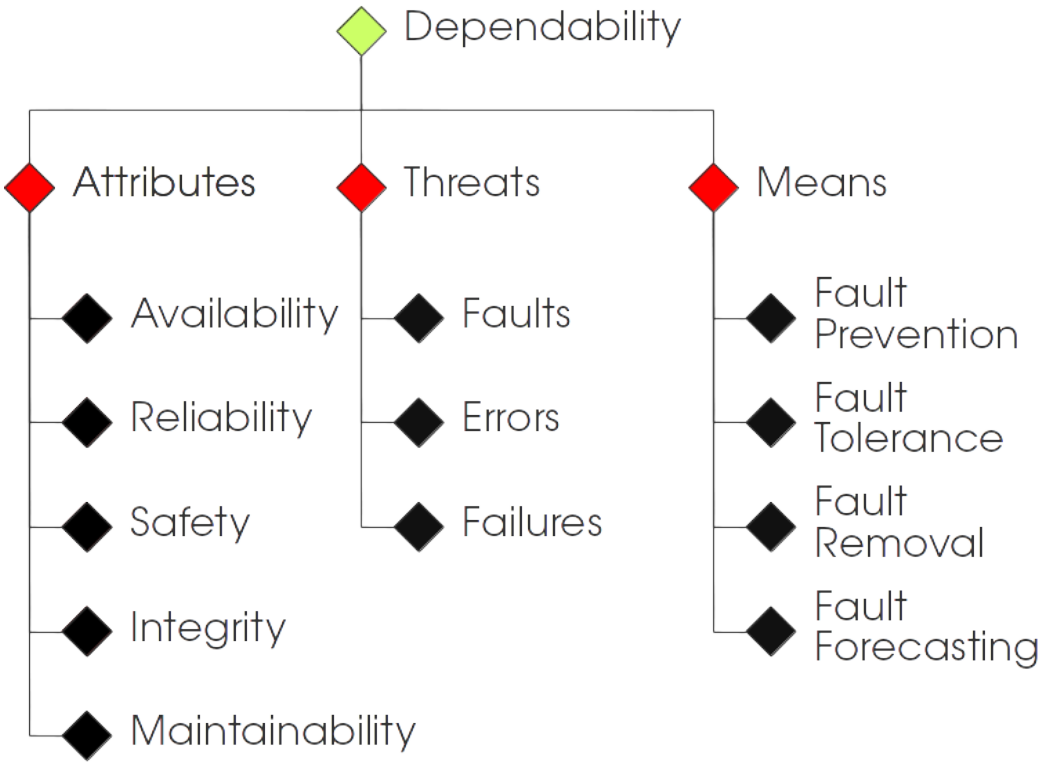


Fig. 2.1 All the aspects of dependability.

Main dependability attributes, as well as dependability threats and means to improve dependability are shown in Figure2.1. A definition of dependability threats is the following:

Failure *an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with functional specification, or because this specification did not adequately describe the system function. Correct service is delivered when the service implements the system function [5]. Failures manifest at user domain.*

Error *part of the total state of the system that may lead to its subsequent service failure. It is important to note that many errors do not reach the system's external state and thus do not cause a failure [5]. Errors manifest at informational domain.*

Fault *adjudged or hypothesized cause of an error. A fault is active when it causes an error; otherwise it is dormant [5]. Faults manifest at the physical domain.*

Among all the attributes of dependability introduced in Figure 2.1, this thesis focuses on reliability. Reliability is the property of a system to behave properly for a given period of time. In particular external agents, manufacturing processes as well as stress can impact reliability of systems, changing their functioning. From a theoretical point of view, reliability is defined as:

Reliability *Reliability is a function expressing the probability of a system to behave correctly, without any failure, in between a certain time interval* $[0,t]$. *Therefore, system reliability,* $R(t)$*, is the probability that the time to failure,* T*, is grater than* t*:*

$$R(t) = P\{T > t\} \qquad where \quad R(0) = 1 \quad \lim_{t\to\infty} R(t) = 0 \tag{2.1}$$

The relation between reliability and failure is highlighted in Equation 2.2, where $F(t)$ is the probability of a failure to happen before time t:

$$F(t) = 1 - R(t) = P\{T < t\}, \quad f(t) = \frac{\mathrm{d}F(t)}{\mathrm{d}t} \tag{2.2}$$

In fact, $F(t)$ expresses the cumulative distribution function (CDF) of the failure distribution $f(t)$, that is the probability distribution function (PDF) of the failures. When taking into account reliability constraints in the design of a system, fault models alongside their failure distribution should be studied at first.

2.2 Reliability metrics of digital systems

Reliability of digital systems can be measured according to different metrics. This section analyzes the main reliability metrics that are widely employed in state-of-the-art studies. In details, each metric can be computed at every stage of the system project, according to the needs of designers.

2.2.1 Failure Rate

The failure rate is the frequency with which the system fails and it expresses the number of failures per unit of time. If the Failure Rate is assumed as constant it can be used to model the exponential distribution of failures and it is commonly indicated as λ. A formal definition of exponential distribution applied in the reliability field is discussed in Section 2.4.

2.2.2 Failures In Time

Digital systems are usually characterized by very low failure rates, consequently another parameter derived from the failure rate is preferred: the Failures in Time (FIT). The FIT is the number of expected failures in a billion hours of system activity.

$$FIT = \lambda_{hours} \times 10^9 \tag{2.3}$$

The most common approach employed to evaluate the reliability of a digital system, expressed by FIT_S, consists of analyzing the FIT of each component of the system, FIT_{C_i}, individually and later combining them together according to:

$$FIT_S = \sum_{i}^{\#components} FIT_{C_i} \tag{2.4}$$

This formula can be employed when errors in each system component i are assumed as independent.

2.2.3 Mean Time To Failure

Mean Time To Failure (MTTF) is the average time before the occurrence of a failure, that is the expected value $E(T)$ of the failure distribution. It is strongly related to the system reliability:

$$MTTF = E(T) = \int_0^\infty t\, f(t)\, dt = \int_0^\infty R(t)\, dt \tag{2.5}$$

The MTTF is usually expressed in hours and it is a basic measure of reliability for non-repairable items. For non-repairable items with constant failure rate:

$$MTTF = \frac{1}{\lambda} \tag{2.6}$$

Finally, MTTF and FIT metrics are correlated as illustrated in Equation 2.7 (a mnemonic rule of the correlation between FIT and MTTF is that an MTTF of 1000 years translates into a FIT rate of 114 FIT):

$$MTTF(years) = \frac{10^9}{FIT \times 24\,hours \times 365\,days} \tag{2.7}$$

2.2.4 Mean Time To Repair

Expresses the mean time to repair an error once it is detected. It therefore measures the service interruption. This time is determined by the repair and recovery mechanisms that a system is equipped with. It is a basic measure of the maintainability of repairable items.

2.2.5 Mean Time Between Failures

The Mean Time Between Failure (MTBF) is the average time elapsing between two failures of a system. MTBF is computed as the sum of two distinct contributions: the Mean Time To Failure (MTTF) and the Mean Time To Repair (MTTR).

$$MTBF = MTTF + MTTR \tag{2.8}$$

In particular, MTTR applies only for repairable systems, systems which include repair mechanisms, in the other cases the MTBF is equal to MTTF.

2.2.6 Availability

Ability of a system to be in state to perform a required function at a given instant of time or at any instant of time within a given time interval, assuming that the external

resources, if required, are provided [6].

$$Availability = \frac{MTTF}{MTTF + MTTR} \tag{2.9}$$

In conclusion, the smaller the MTTR the higher is the availability of the system.

2.2.7 Mean Workload To Failure

It captures the average amount of work between two errors and is useful to compare the reliability of different workloads [7].

2.2.8 Mean Instruction To Failure

It expresses the average number of committed instructions in a microprocessor between two errors [8].

2.3 Dependability threats

Dependability threats are necessary to understand and identify the weak points of the system. As explained in Section 2.1 the dependability threats are generally described by the concepts of faults, errors and failures (see Figure 2.2). Metrics introduced in Section 2.2 related to dependability threats measure the probability of occurrence of these events as well as the relations among these events.

2.3.1 Faults

Following the taxonomy of [5], the faults can be classified according to different properties: the phase of creation or occurrence, the system boundaries, the phenomenological cause, the dimension, the objective, the intent, the capability and the persistence (Figure 2.3). Moreover, the concept of intermittent faults [9] has been added to the used taxonomy in order to take into account the different nature and impact of these faults.

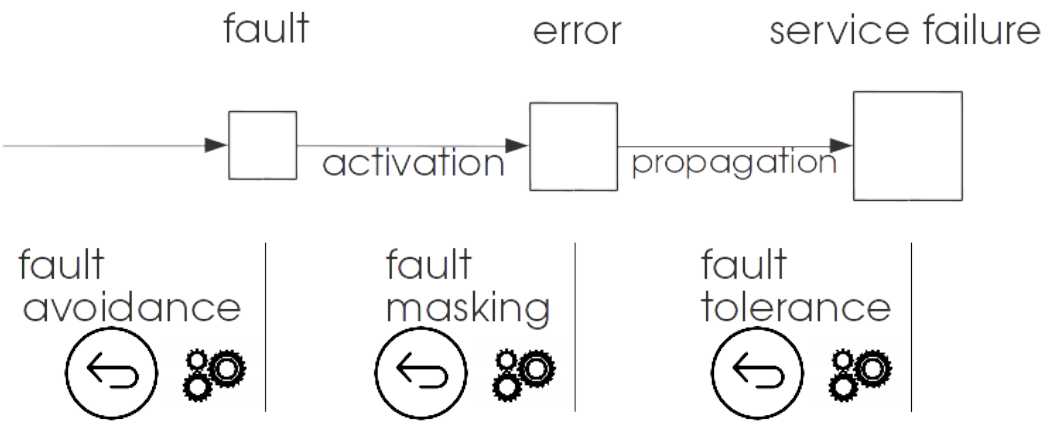


Fig. 2.2 The chain of dependability threats.

In the context of this thesis, only hardware faults are considered. Thus, the types of faults that are covered are the following:

- Manufacturing defects: open or short circuits, parametric failures
- Physical deterioration: wear-out effects like NBTI, electromigration, TDDB, HCI ...
- Physical interference: soft errors and electromagnetic interferences (EMI)

In literature, hardware faults are usually classified depending on their persistence, other than their natual causes. According to Figure 2.3 there are three categories of fault persistence:

- **Transient faults**: they have a limited duration and can be recovered by rewriting the affected resource or by a physical reset. There are different kinds of transient faults, based on the affected resources:
  - **Single Event Upset (SEU)**: it is a bit-flip in a memory element. The occurrence of SEUs depends on technology and environment, soft errors in particular.
  - **Multiple Bit Upset (MBU)**: it is very similar to the SEU, but it affects multiple bits. A MBU is defined as “any event or series of events that cause more than one bit to be upset during a single measurement” [10].

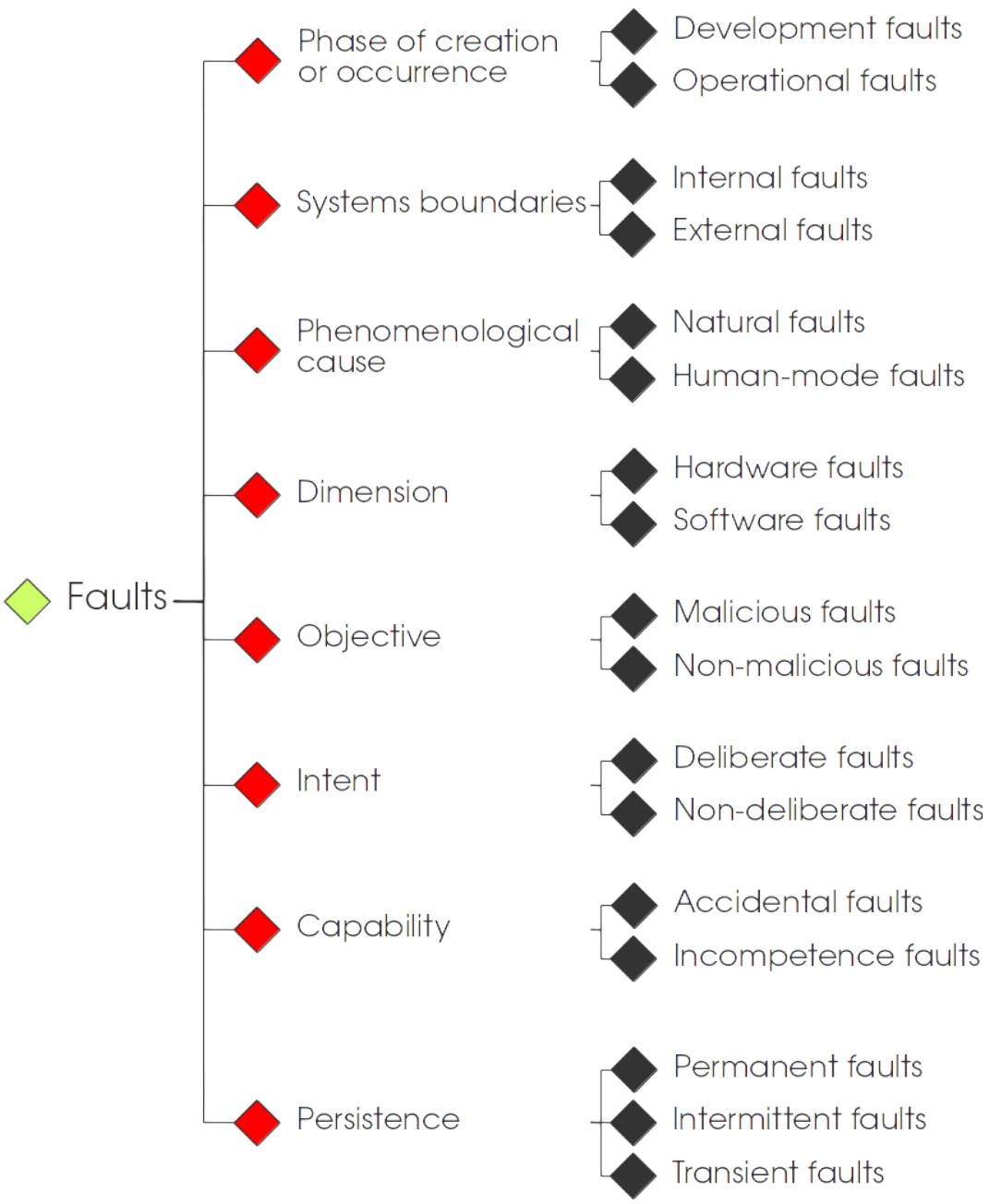


Fig. 2.3 Classification of faults.

– **Single Event Transient**: it is defined as a "momentary voltage excursion (voltage spike) at a node in an integrated circuit caused by a single energetic particle strike" [11] affecting logic gates behavior.

  - **Single Event functional Interrupt**: it is "a soft error that causes the component to reset, lock-up, or otherwise malfunction in a detectable way, but does not require power cycling of the device (off and back on) to restore operability" [11].
  - **Single Event Latch-Up**: it is a "abnormal high-current state in a device caused by the passage of a single energetic particle through sensitive regions of the device structure and resulting in the loss of device functionality" [11].

- **Permanent faults**: they have unlimited duration and their origin is attributed to manufacturing defects and physical deterioration. Once a resource is affected by a permanent fault, it is impossible to restore operability of such a resource.

- **Intermittent faults**: they are due to physical deterioration or residual defects, not detected during manufacturing test, and activated only under some particular conditions. They are in the middle between permanent and transient faults. "After their first appearance, they usually exhibit a relatively high occurrence rate and, eventually, tend to become permanent" [12]. In detail, the most common intermittent faults are delay faults, these faults are provoked by wear-out which alters the impedance of the circuits and consequently causes a delay in the propagation of electrical signals. Even if a delay fault is present it does not manifest all the times, since the delayed signal does not always influence the output. Recovery from intermittent faults can be managed resorting to a change of circuit operating parameters such as voltage and frequency.

2.3.2 Errors

When faults are activated, they result in errors. However, a large part of faults are not activated but dropped. The possible outcomes of a single-bit fault in different states have been classified in [13] and are represented in Figure 2.4. More specifically, as a fault manifests it can be immediately detected and corrected by proper mechanisms. In case neither correction nor detection are implemented the fault turns into an error and it is classified as Silent Data Corruption (SDC). Conversely, if only detection mechanism is present the fault is classified as Detectable Unrecoverable Error (DUE),

this error category is divided into two types: False DUE, in case the error does not affect the application outcome, and True DUE, in the opposite case.

The most common metrics employed for errors address both the probability of occurrence of errors and the probability of activation of faults resulting into an error. They are:

- **SDC rate**: it is the probability of occurrence of a fault to manifest as SDC. [14]
- **DUE rate**: it is the probability of faults that are detected but cannot be recovered.
- **Architectural Vulnerability Factor (AVF)**: it measures the vulnerability of a hardware structure to faults. AVF is defined as the probability that a fault in particular structure will result in an error (i.e. SDC and DUE) [15].
- **Program Vulnerability Factor (PVF)**: Captures the architecture-level fault masking inherent in a program, allowing software designers to make quantitative statements about a program's tolerance to faults [16].
- **Hardware Vulnerability Factor (HVF)**: Quantifies the vulnerability of hardware structures to errors [16].

2.3.3 Failures

A failure of the system happens when the delivered service deviates from the correct service. The way the service deviates from its correct execution is the failure mode of the service. Figure 2.5 represents the set of failure modes of services proper of a computing system, as defined in [5]. Beside its mode, a failure is characterized by:

- **detectability**: if the failure of the service can be detected and signaled to the user;
- **consistency**: if the failure is perceived identically by all system users;
- **severity**: if the failure has a minor impact or catastrophic consequences. The notion of failure severity is heavily dependent on the application domain. There are no generic definitions of the severity of the system.

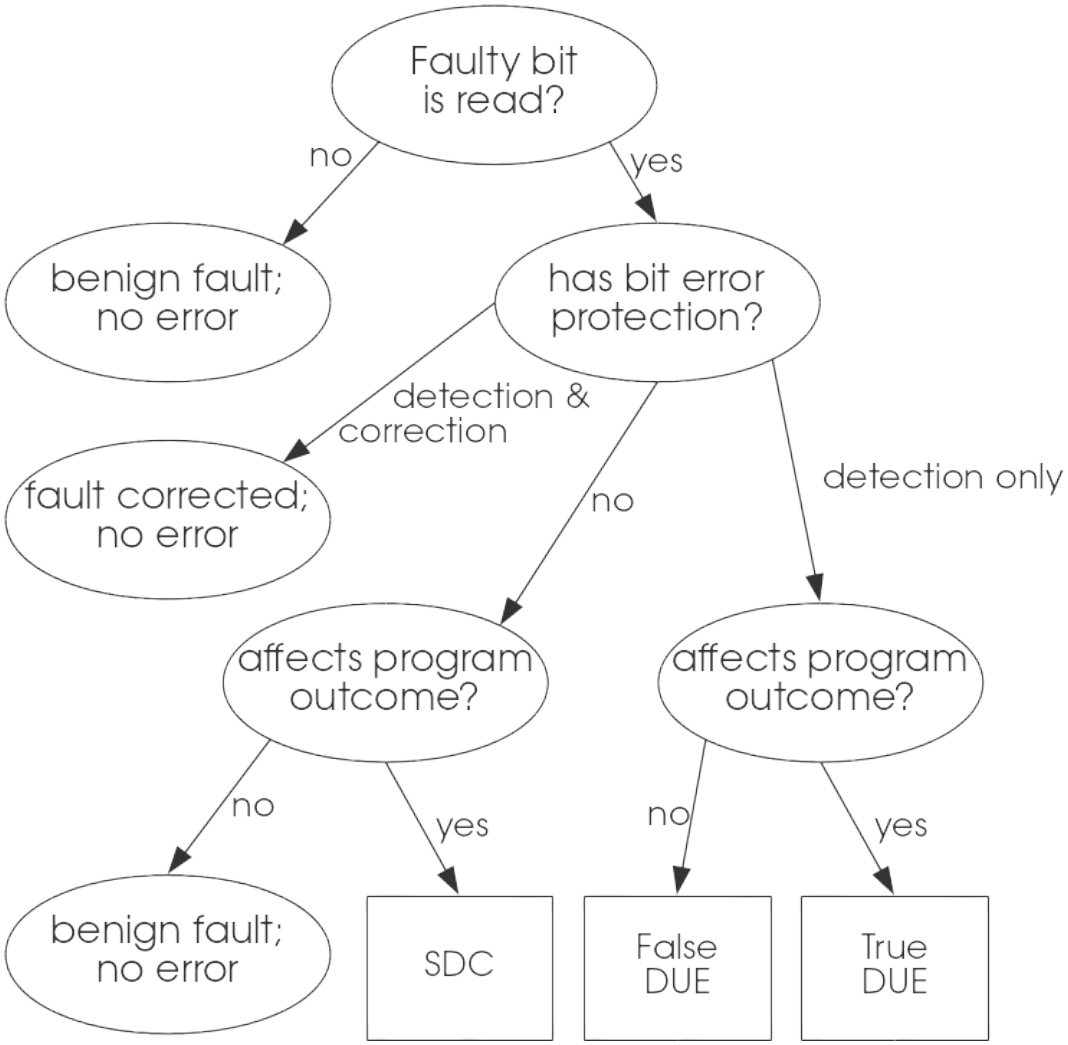


Fig. 2.4 Classification of errors caused by faulty bits.

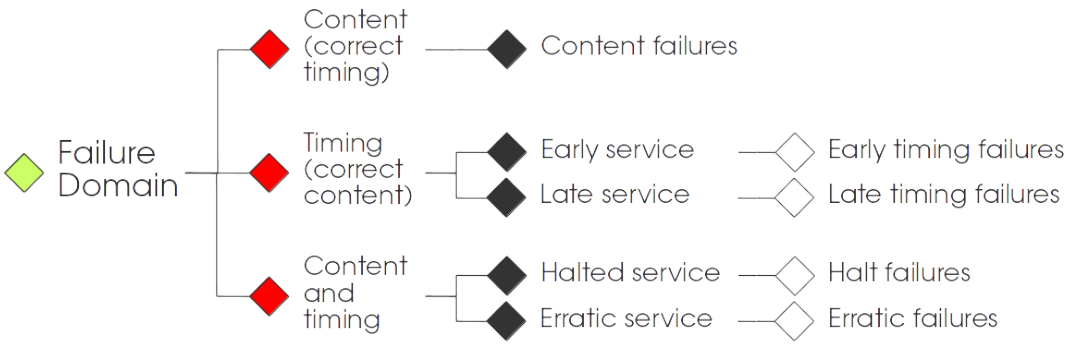


Fig. 2.5 Failure modes of computing systems.

Metrics on failures for software application of digital systems

This thesis mainly focuses on microprocessor-based digital systems. For this reason, this subsection briefly introduces their main failure metrics.

- **Early timing failure rate**: Probability of early timing failure.
- **Late timing failure rate**: Probability of late timing failure.
- **Deadline miss ratio**: Percentage of task deadlines missed in soft real-time applications.
- **Fatal Hardware Traps rate**: Probability of Fatal Hardware Traps.
- **Hang rate**: Probability of failure due to a hang in the application or OS.
- **Abnormal application exit rate**: Probability of Abnormal Application Exit.
- **High OS Activity rate**: Probability of failure due to High OS activity.

2.4 Soft errors

This thesis focuses on soft errors are errors activated by Radiation Induced Faults (RIF)[14] [17]. According to Figure 2.3, based on [5], RIF are operational faults generated internally. These faults are accidental, non-malicious and non-deliberate. RIF affect the hardware of the system and their effects are transitory. The causes of RIF are natural, in detail, they can be produced due to different types of sources: alpha particles, from packaging and neutrons, from the atmosphere. Radiation faults are addressed with fault detection and error correction circuitry. Soft error caused by intentional perturbations (attacks) are out of the scope of this work.

An alpha particle consists of two protons and two neutrons bound together into a particle. Alpha particles are emitted by radioactive nuclei, such as uranium or radium, in a process known as alpha decay. Alpha particles have kinetic energies of a few MeV, which is lower than those of neutrons that affect CMOS chips. Nevertheless, alpha particles can affect semiconductor devices because they deposit dense track of charge and create electron-hole pairs as they pass through the substrate. Alpha

particles can arise from radioactive impurities used in chip packaging such in the solder balls or contamination of semiconductor processing materials. Alpha particles are difficult to eliminate completely from the chip so chips need fault detection and error correction techniques.

The neutron is one of the subatomic particles that make up an atom. Atoms are considered the basic building blocks of matter and consists of three types of subatomic particles: protons, neutrons and electrons. A proton is positively charged, a neutron is neutral and an electron is negatively charged. An atom consists of an equal number of protons and electrons and hence it is neutral itself. The neutrons that cause soft errors arise when atoms break apart into protons, electrons and neutrons. Protons have a long half-life so can persist for longtime before decaying and constitute the majority of the primary cosmic rays that bombard the earth's outer atmosphere. When these protons and associated particles hit atmospheric atoms, they create a shower of secondary particles named secondary cosmic rays. Untimely, the particles that hit the earth's surface are known as terrestrial cosmic rays.

One key concept to explain the interaction of alpha particles with silicon is the stopping power. Stopping power is defined as the energy lost per unit track length, which measures the energy exchanged between an incoming particle and electrons in a medium. Stopping power quantifies the energy released from an interaction between radiations (alpha particles and neutrons) and silicon crystals, which in turn can generate electron-hole pairs. About 3.6 eV of energy is required to create one such pair. Whether the generated charge can actually cause a malfunction or a bit flip depends on two factors named charge collection efficiency and critical charge of the circuit that will be explained later.

Both alpha particles and neutrons can produce an electrical charge. When this charge is sufficient to overwhelm a circuit, then it may malfunction. At the gate or cell level, this malfunction appears as a bit flip. For storage devices, when a bit residing in a storage cell flips, a transient fault is said to have occurred. In contrast, for logic devices, a change in the value of the input node feeding a gate or output node coming out of a gate does not necessarily mean a transient fault has occurred.

In fact, a fault occurs only when this fault propagates to a forward latch or storage cell.

This minimum charge necessary to cause a circuit malfunction is termed as the critical charge of the circuit represented as *Qcrit*. Typically, *Qcrit* is estimated in circuit models by injecting different current pulses till the circuit malfunctions. Once *Qcrit* is defined, the Soft Error Rate (SER) of a digital circuit can be computed according to physical models. Hazucha and Svensson [18] proposed a model to predict neutron induced SER:

$$Circuit\,SER = k \times Flux \times Area \times e^{-\frac{Qcriti}{Qcoll}} \tag{2.10}$$

Constant is a constant parameter dependent on the process technology and circuit design style, $Flux$ is the flux of neutrons at the specific location, Area is the area of the circuit sensitive to soft errors, and *Qcoll* is the charge collection efficiency, which is the ratio of collected and generated charge per unit volume. *Qcoll* depends strongly on doping and Vcc and is directly related to the stopping power, so the greater is the stopping power, the greater is *Qcoll*. *Qcoll* can be derived empirically using either accelerated neutron tests or device physics models, whereas *Qcrit* is derived using circuit simulators. Finally, the impact of soft errors on future is reported in Appendix A.

As anticipated in Section 2.2.1, exponential distribution is employed when a constant failure rate, λ, is assumed. This is what happens for soft errors of digital systems where the failure rate corresponds to the SER. In detail, the SER does not change over the time, but it is mainly related to environmental parameters. On the opposite, when considering wear out, the failure rate changes with time, as a consequence, the exponential distribution cannot be used anymore.

From a theoretical point of view, the probability density function of a random variable x and parameter λ with an exponential distribution is defined as:

$$f(x) = \lambda e^{-\lambda x}, x \geq 0 \qquad f(x) = 0, x < 0 \tag{2.11}$$

The exponential cumulative distribution function can be computed as:

$$F(x) = \int_{-\infty}^{x} \lambda e^{-\lambda x} dx = 1 - e^{-\lambda x} \Rightarrow R(t) = e^{-\lambda x} \tag{2.12}$$

MTTF can be easily derived for exponential distribution from equation 2.5:

$$MTTF = \int_0^{\infty} R(t)\, dt = \int_0^{\infty} e^{-\lambda t}\, dt = \left| -\frac{e^{-\lambda t}}{\lambda} \right|_0^{\infty} = \frac{1}{\lambda} \tag{2.13}$$

This result highlights the relation between the MTTF and the parameter λ, which is the failure rate.
The most important property of the exponential distribution is the absence of memory about what has happened in the past. In details, if the reliability function is computed for T_0 assuming the system is working properly at time T_0 the reliability function is not altered:

$$R(t|T_0) = \frac{P(T > t + T_0)}{P(T > T_0)} = \frac{e^{-\lambda (t+T_0)}}{e^{-\lambda T_0}} = e^{-\lambda t} = R(t) \tag{2.14}$$

This result underlines that the occurrence of a soft error does not influence the time next error will appear.

2.5 Dependability enhancement techniques

This Section aims at introducing the state-of-the-art techniques employed to improve the reliability of computing systems in presence of soft errors. A more detailed analysis can be found in [19], here only the most relevant aspects are reported.

Modern and future systems are demanding more and more computational capabilities, however, satisfying reliability of these systems represents a fundamental requirement to achieve exascale performance [20]. Consequently, the designers should consider reliability as one of first-order design constraints. This challenge has attracted researchers attention in recent years and several solutions to improve reliability of computing systems have been investigated.

Several approaches can be employed to improve dependability of a system implementing different techniques to achieve error detection, diagnosis and recovery

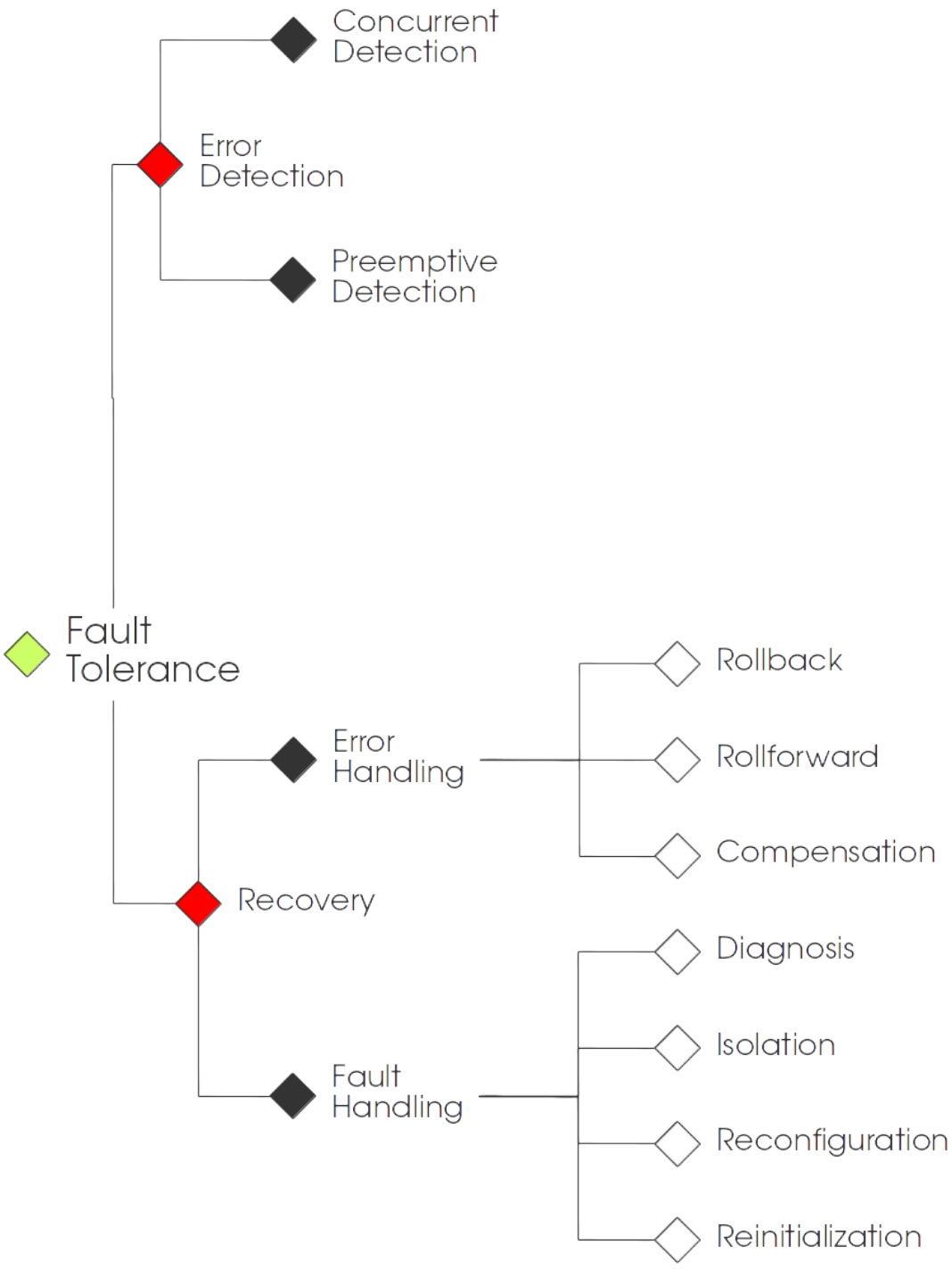


Fig. 2.6 The dependability enhancing mechanisms.

(Figure 2.6). The majority of current systems concentrate their reliability mecha-

nisms at circuit and architecture level. In detail, techniques to improve reliability of microarchitectural components target registers, functional units, cache and main memories. These mechanisms involve one or two layers of the system stack (Figure 2.7), while the other layers of the system assume that the hardware operates reliably and predictably since all the errors manifesting at hardware level are always corrected. The main drawback of this approach is that worst-case scenarios are assumed when designing protection mechanisms because of the lack of system-wide information. As a result, convenience is emphasized over efficiency: more and more protection is required to guarantee correct operational conditions alongside the constant increase of failure rates due to soft errors, variation and aging of circuits as feature sizes shrink.

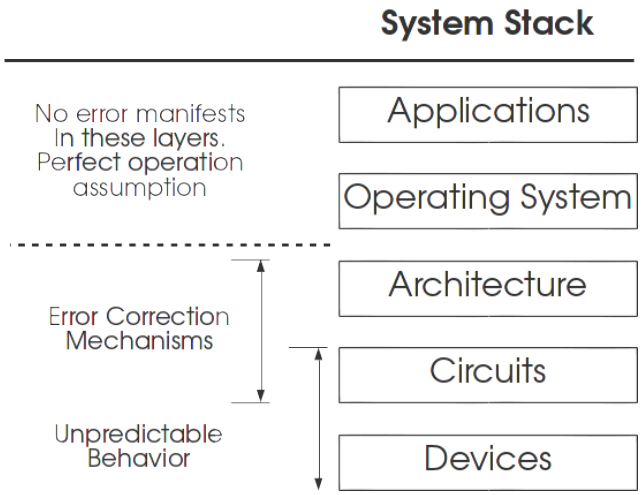


Fig. 2.7 The conventional approach to enhance reliability usually targets one or two layers of the system.

2.5.1 The traditional approach

Nowadays, in scenarios such as aerospace, avionics and nuclear plants, where reliability requirements are high, the typical solutions are based on resource replication. To tolerate hardware errors, several copies of the same resource are computed in parallel and later compared to check output consistency.
Several approaches were studied in literature targeting replication at different system layers. Lyons et al. proposed Triple Modular Redundancy (TMR) in [21], this technique was employed in the Fly-By-Wire mechanism of Boeing 777 [22]. TMR

consists of three copies of the same hardware circuit sharing the input data, while the output data is connected to a majority voter which is in charge to select the final output. A similar approach was proposed at software level [23–26], where multiple copies of software are executed on the same processor and later checked to detect errors and correct it if required.
These techniques are very easy to implement and are not perceived by application software, so that no additional efforts are required when designing new applications. Despite these advantages, they are inefficient if adopted to modern systems. More specifically, they are inefficient from an energy consumption perspective, since duplicating computations to detect errors needs more than twice the energy, as comparing the results of the two computations costs energy too. Having three and more copies means increasing accordingly energy overheads due to multiple copies and more complex voting mechanisms. Moreover, performance is also affected since resources involved in the execution of the copies could be employed to carry out other tasks and the voting introduces an additional time overhead.

Some other redundancy-based techniques were investigated in order to limit the drawbacks of resource replication. Concerning protection mechanisms for L1 caches Kim and Somani propose to employ dedicated error protection circuits for only the most frequently accesses cache blocks [27].
Zhang et al. [28] propose to hold replicas of the active cache block in cache blocks that are not accessed for long time period. In particular, they suggest two replication algorithms. The first consists of replicating a block when the block must be written and a cache miss happens. The second tries to replicate blocks every time they are written.
Zhang [29] instroduces a small fully associative cache (R-cache) to store replica of every block that is written in L1 data cache. For L1 chaches, thanks to high temporal locality, only a few blocks of R-cache can accommodate data for almost all read hits. However for L2 caches, characterized by lower temporal locality, the size of R-cache would be very large, thus resulting in high dynamic energy overheads.
Sugihara et al. [30] propose a reliable architecture for caches based on reliable cache ways composed of two or three cache ways holding the same data, enabling respectively error-detection and error-correction mechanism in spite of a reduction of cache capacity to 50% and 33 %.
A similar approach is presented in [31, 32] where unused registers are used to hold

replicas of active registers. These techniques are usually characterized by negligible reduction of register file capacity. However, some performance overheads are introduced due to error-detection process.
Finally, Wells et al. [33] propose the possibility to limit redundancy for only high reliable applications. More specifically, dual mode redundancy, enabling resource duplication, is disabled for performance-oriented tasks, thus limiting performance overheads.

Low efficiency and high costs of replication have led to the adoption of different mechanisms for systems that can tolerate slightly higher error rates. In particular, lower-costs solutions were studied for large memory arrays, such as error-correcting codes [34, 35], and for datapaths, as residual arithmetic [36]. The adoption of these solutions has grown alongside the error rates in order to guarantee constant reliability performance. Unfortunately, for recent technologies the error rate and the number of physical phenomena producing errors have increased so much that meeting reliability requirements is becoming economically unsustainable.

Worst-case assumption are also employed to tolerate process variation and aging effects. Margining is the most widely-used technique consisting of testing the maximum operating frequency at a given supply voltage of a device and then make the device operate at lower clock frequency and higher supply voltage. In this way, aging and adverse environment effects are mitigated. This approach wastes power and performance and it is suitable in case intra-die device process variation and aging are small enough to justify the margining drawbacks in favor of a simplified design and low cost. However, this was the case of older fabrication processes. On the opposite, as the feature size shrinks, margining is not a cost-effective solution anymore due to the growth of intra-die variation and aging effects.

To conclude, state-of-the-art techniques to enhance reliability lead to inefficient design, characterized by high energy consumption, poor performance and high costs. Industry needs new approaches to cope with soft errors, variation and aging, making technology shrinking profitable and more efficient. The keys to the solution that are currently under investigation identified a common point: a *cross-layer* approach.

2.5.2 Cross-layer reliability enhancement

Traditional solutions to improve system reliability have become too expensive to copy with the challenges of future technologies. The main reason of their inadequacy is that they operate at individual layers, without any information or assistance of the other layers of the system stack, thus introducing burdensome overheads. The key point introduced by cross-layer reliability design is to let cooperate all the layers of the system stack in order to mitigate errors that are not managed efficiently at device and circuit level. An accurate and deep analysis of the advantages that a cross-layer reliability approach can enable is presented in [37]. Here the key aspects described by this document are reported.

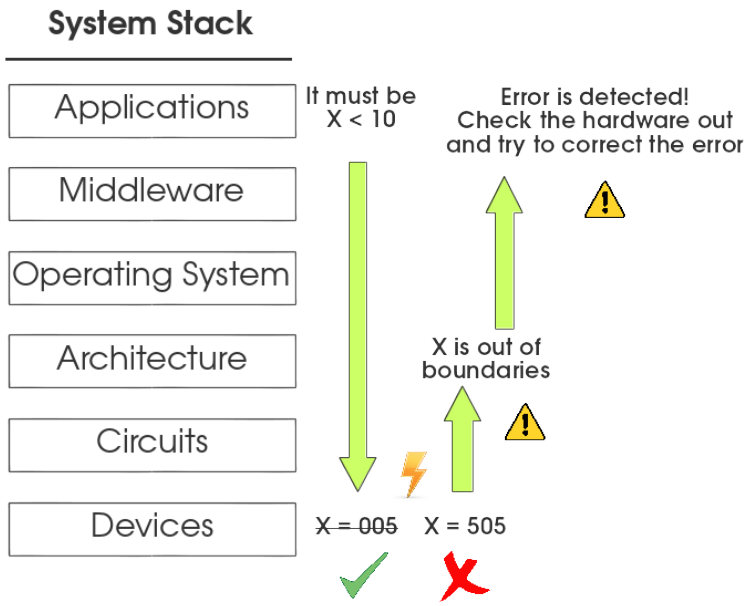


Fig. 2.8 Cross-layer information sharing and cooperation.

The cross-layer approach is shown in Figure 2.8 by an example. Supposing that the application layer determines a bound on the value of a variable x, this information is then shared among all the system layers, the opposite of what happens for conventional systems. In the case the architecture layer detects a misbehavior, that is x out of bounds, this information is again shared among all the system layers. More specifically, the error is not immediately corrected by the layer that has detected it, but it can signal the middleware and OS layers to contain and correct the error before it becomes visible to the application. Indeed, the fundamental aspect of the cross-

layer approach is the sharing and the cooperation of all the system stack, requiring a radical change in the way computing systems are designed and engineered.

Some aspects related to the components of a computing system can be exploited to move towards a cross-layer perspective:

- Hardware and software should be designed for repair. Computational organizations should be prepared for error detection and a dynamic reconfiguration to overcome error occurrence. In this context, mitigation of errors can be managed by the cooperation across architecture and operating system.
- The capability of filtering error should be possible at different levels from circuits through software applications. In a scenario where some circuits allow errors to propagate to the other layers in which detection and correction mechanisms can take place, multilevel trade-offs between hardware and software protections can be investigated to provide efficient solutions.
- Lightweight error checking mechanisms exploiting properties of software applications and algorithms have the potential to increase efficiency letting hardware operate safely on the edge of failure, avoiding energy consumption overheads due to margining.
- Adapting system error rates to application reliability demands would increase efficiency by tailoring circuit properties and hardware resources. This solution requires a computing system composed of different components with the same functionality but different implementation and features (i.e., reliability, power consumption and performance).

Chapter 3

Fault tolerance in autonomous FPGA-based systems

This thesis mainly focuses on reliability evaluation of systems composed of off-the-shelf components. However, this Chapter provides a methodology to enhance reliability of FPGA-based systems. Reliability evaluation and fault tolerant design are difficult tasks when off-the-shelf components are employed. For this reason, Field-Programmable Gate Arrays (FPGAs) are becoming widely employed in critical applications, where lifetime and system dependability must be improved since they represent a neuralgic point of the system. This chapter addresses the study of a methodology to increase fault tolerance in autonomous FPGA-based systems. Part of this chapter was previously published in [1]. More specifically, the presented methodology aims at protecting the system from both permanent and transient faults by means of Dynamic Partial Reconfiguration (DPR) of the FPGA. DPR is employed to correct faults exploiting reconfiguration of faulty modules at run-time, without stopping the FPGA modules which are working correctly. A design flow and a partitioning strategy to maximize the number of permanent faults the system can tolerate are proposed.

3.1 Introduction

FPGA are increasingly employed in many application domains, ranging from ASICs (Application Specific Integrated Circuits) prototyping, datacenters [38] to embedded

systems in critical scenarios (i.e., automotive [39] and aerospace [40]). Safety-critical scenarios, in particular, require high reliability, availability and lifetime. FPGAs are well suited to deliver the needed dependability requirements thanks to their high flexibility, reconfigurable characteristics and limited non recurrent engineering cost.

FPGAs are reprogrammable circuits composed of different kinds of blocks connected by reconfigurable wires. The blocks an FPGA is composed of can be: logic blocks, where logic gates are accommodate, RAM blocks (BRAM) containing memory arrays, and other types of block which are specific of FPGA models. In order to work correctly, the device must be configured according to the tasks required by the running applications. More specifically, the FPGA provides a configuration memory which is written with the appropriate configuration file, named bitstream. In recent years, DPR was introduced [41], enabling the possibility of reconfigure just a portion of circuit while the reminder of the device continues to work properly.

An effective system maintenance can be carried out by means of DPR remotely, on-site or at run-time. In the case availability is a top concern, remote maintenance is not always possible and, in addition, it is a slow process. Autonomous Fault-Tolerant Systems (AFTS) can overcome availability issues leveraging their autonomous capability of self-recovery, offering longer lifetime and better availability [42].

The most popular FPGAs are SRAM-based, that is, their configuration memory is an SRAM memory. Both transient and permanent faults can affect FPGAs. More specifically, soft errors can provoke Single Event Upsets (SEUs) and Multiple Bit Upsets (MBUs). As described in Section 2.4, soft errors are caused by radiations and their effects are temporary [43, 44], while aging and wear-out induce permanent faults. Several works addressing mechanisms to enhance reliability in presence of soft errors were presented in the literature [45] [46], while protection mechanisms against permanent faults have not been deeply investigated [47] [48] despite their importance as highlighted by the International Road Map for Semiconductors [49]. In detail, as the feature sizes of digital circuits are increasingly shrunk, the rate of permanent faults increases too. In addition, in some critical applications, the lifetime of the systems is expected to be of several years. As a result aging and wear-out effects become not negligible and must be considered during the design process.

The goal of this chapter is to introduce a design methodology to enhance reliability for permanent and transient faults by means of an autonomous fault tolerance mechanism leveraging FPGAs. DPR is at the basis of this protection technique,

enabling fast recover, increased availability and longer lifetime. The related design flow and partitioning strategy to maximize the number of tolerated permanent faults are described later in this chapter. The main advantages introduced by this work with respect to state-of-the-art solutions is a dramatic reduction of the recovery time and memory space required to store recovery information.
In order to give the reader a complete view of the problem under study, at first, solutions proposed in literature are introduced, then, the proposed methodology and the results gathered from two case studies are presented. Finally conclusions are discussed.

3.2 State-of-the-art

Regarding FPGAs, while detection of permanent faults was the object of several studies in literature [50, 51], only few solutions were proposed for fault tolerance enhancement and fault recovery. Two approaches can be adopted to improve reliability against permanent faults.

The first is based on redundancy and its main advantage is that the system continues to operate correctly in presence of faults, without any interruption and associated performance loss. However, this approach introduces a large hardware overhead in terms of area and power consumption. Triple Modular Redundancy (TMR) is the most popular technique based on redundancy [52]. It is characterized by the triplication of the same hardware functionality and a majority voter, computing the final output by comparing the results of the triplicated circuits, thus resulting in an area overhead more than three times higher.

The second is based on DPR, leveraging the reconfiguration of faulty hardware modules. This approach is more efficient under several aspects: it introduces an area overhead smaller than TMR, considering the same number of tolerated permanent faults, and it enables autonomous fault recovery, thus obtaining Autonomous Fault-Tolerant Systems [42]. Exploiting reconfiguration to manage permanent fault in SRAM-based FPGA systems was discussed in [53–56]. When electromigration and wear-out effects manifest as permanent faults [57], the main idea is to identify and delimit the corrupted area. Successively, the portion of circuitry affected by the fault is relocated. The relocation process consists of configuring a portion of the FPGA with a functionality identical to the one of the faulty part and preventing the usage of

the permanently damaged area. In order to relocate a hardware module, the proper bitstream (configuration file) must be loaded into the configuration memory of the FPGA. In detail, when resorting to DPR, it is possible to configure just a portion of the FPGA, instead of configuring the entire device. Moreover, the portions of FPGA that are not involved in the configuration process continue to work without being stopped.

Recovery methodologies targeting both transient and permanent faults are presented in [47, 58, 48]. In [47] and [58] authors propose a methodology to recover from both permanent and transient faults. Whenever a permanent fault is detected, the proposed recovery strategy consists of reconfiguring the FPGA with a pre-computed bitstream which does not use the faulty FPGA resources. However, these works do not exploit DPR because every time a new bitstream must be loaded, the FPGA is reconfigured entirely, thus increasing the recovery time. In addition, this solution requires a lot of memory space to store all possible bitstreams since a single bitstream for the whole FPGA is composed of several MB of data and the number of pre-computed bitstreams grows exponentially with respect to the number of partitions the system is composed of.
The methodology proposed in [48], instead, exploits DPR. The paper proposes a pipelined architecture of the circuit to be implemented. In this case, the usage of partial reconfiguration slightly reduces recovery time and memory requirements for bitstreams storage. However, because of the pipelined architecture, the mechanism to recover from permanent faults suffers from a slow and inefficient recovery strategy, as the faulty pipeline stage as well as the following stages must be reconfigured. Moreover, the proposed methodology does not address faults affecting interconnections among adjacent pipeline stages, thus introducing a limitation in its adoption to real use-cases.

The methodology described in this chapter aims at overcoming the limitations of state-of-the-art solutions by exploiting DPR to relocate efficiently faulty modules of the FPGA at run-time. More specifically, recovery time is reduced leveraging partial reconfiguration and, consequently, reducing dramatically the memory to store bitstreams. Moreover, spare hardware resources of the FPGA are dedicated to recovery from permanent faults affecting interconnections among functional modules. The methodology consists of a design flow and a partitioning strategy to maximize the number of tolerated faults employing spare resources of the FPGA. As a result, availability and lifetime of FPGA-based AFTS is improved.

3.3 The proposed methodology

The proposed methodology consists of an FPGA-based system architecture, a partitioning strategy and a partitioning algorithm in order to offer efficient recovery. The system architecture and the partitioning methodology exploits the advantages introduced by DPR [41], while the partitioning algorithm is designed to find the best partitioning in terms of tolerated permanent faults.

3.3.1 The proposed system architecture

The proposed architecture targets AFTSs implemented on SRAM-based FPGAs. In order to autonomously detect and recover from faults, the architecture of the designed system is divided into three blocks, carring out different tasks (Figure 3.1). The three blocks are:

- the *Application FPGA*: a SRAM-based FPGA implementing the functionalities required by the system and partitioned into tiles;
- the *Fault Manager*: in charge of classifying the detected faults (*Fault Classifier*) and managing the reconfiguration process of the *Application FPGA* (*Configuration Controller*);
- the *Bitstream memory*: storing all the configuration files related to the *Application FPGA*.

The main component of the architecture is the *Fault Manager* which monitors faults occurring in the *Application FPGA* and manages the recovery process. More specifically, the monitor process is performed by the *Fault Classifier* and it consists of two task: identifying which tiles are faulty and whether the fault is transient or permanent. This can be achieved either running periodic tests on the *Application FPGA*, as described in [50, 51], or including error detection mechanisms in the tiles of the *Application FPGA*. Every detected error is signaled to the *Fault Classifier* [59, 60].
The recovery is managed by the *Configuration Controller*. It runs the proper recovery operations on the basis of the fault classification so that the system is restored back to work. In particular, the recovery operations from a permanent fault occurring in

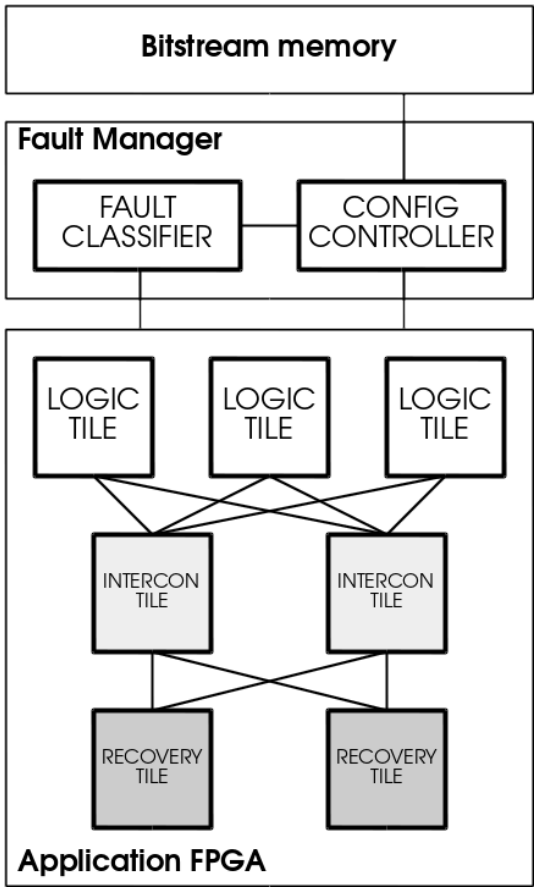


Fig. 3.1 The proposed system architecture (Source: [1]).

the *Application FPGA* consists of a reconfiguration of the FPGA. For this reason the *Configuration Controller* is connected to the *Bitstream memory*, thus guaranteeing a direct access to configurations.
Since the role of the *Fault Manager* is of primary importance it can be implemented resorting to fault tolerance techniques [61, 62]. Moreover, to avoid errors in the configuration files, the *Bitstream memory* can embed error detection and correction codes [63]. However, the actual implementation of the *Fault Manager* and the *Bitstream memory* are not addressed in this work and are assumed to be fault-free.

Finally, the *Application FPGA* hosts the hardware modules required to implement the system functionalities. Its area is divided into several partitions named *tiles*. The following sections presents the main features of the *tiles* as well as the partitioning strategy adopted to maximize the number of permanent faults that can be tolerated.

3.3.2 The proposed partitioning methodology

The *Application FPGA* is composed of three different types of *tiles*, characterized by different roles in the system design, as illustrated in Figure 3.1. More specifically, the computation and the data processing is the task of *logic tiles*, while communication among logic tiles is guaranteed by *interconnection tiles*. Finally, the remaining part of the FPGA is divided into *recovery tiles* which are used as spare resources. In fact, every time a permanent fault occurs in a logic tile, the faulty tile is relocated into a recovery tile.

In order to employ the proposed architecture, minimal effort is needed during the design phase. In fact, the implementation of the hardware functionalities must be built with a modular approach. The system is divided into basic components, interconnected to each other. Later, basic components are organized and grouped together into logic tiles according to the amount of required FPGA resources (i.e., slices, BRAMs and DSPs [64]) and the number of connections to the other basic components (Figure 3.2). A logic tile is characterized by a certain amount of resources and interconnections, therefore basic components can be accommodated into logic tiles if their resources needs are satisfied.

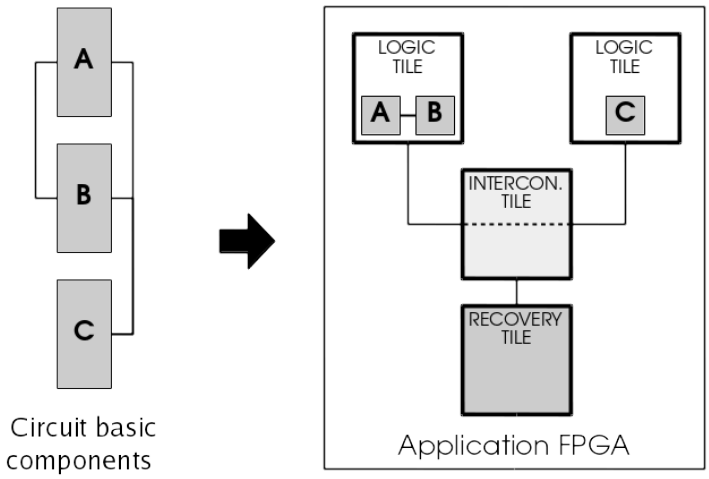


Fig. 3.2 Circuit basic components and tiles organization (Source: [1]).

The fault recovery strategy in presence of transient faults is the same adopted in [58]. When an error in a tile is signaled to the Fault Classifier for the first time, the cause of the fault is attributed to a transient fault in the circuit of the faulty tile. For this reason the application is reset and re-executed. In case the error appears

again after the first recovery stage, the Fault Classifier assumes a transient fault in the configuration memory, so the bitstream of the faulty tile is re-written into the configuration memory of the *Application FPGA* by the Configuration Controller. If the fault appears after a pre-specified sequence of detection-recovery operations, then, it is classified as permanent.

The recovery mechanism in presence of permanent faults in a logic tile consists of moving the faulty tile to a spare recovery tile. However, the recovery tile chosen for the relocation must guarantee a sufficient number of resources to host the hardware resources of the faulty logic tile. In the proposed partitioning strategy, all the recovery tiles have a number of resources equal to the one of the most demanding logic tiles. This solution allows to relocate each logic tile into each recovery tile, therefore the number of recovery tiles is equal to the maximum number of faulty logic tiles the system can tolerate.

When a permanent fault in a logic tile is detected, the *Configuration Controller* loads the proper configuration file into the FPGA configuration memory to relocate the faulty tile into a recovery tile (Figures 3.3a and 3.3b). However, this operation is not sufficient as the interconnections have to be updated as well. To overcome this problem, the active interconnection tile is also reconfigured by the *Configuration Controller*, so a new configuration file for the interconnections tile is loaded (Figure3.3b).

Backup interconnection tiles are added to the design to cope with permanent faults affecting interconnections (Figure 3.3c). In detail, one and only one interconnection tile is active at a time, while the remaining ones act as spare interconnection tiles and are properly activated when faults occur. The number of required backup interconnection tiles must be equal to the number of interconnection faults (i.e., permanent faults affecting interconnection tiles) the system must tolerate. Whenever a permanent fault occurs in the active interconnection tile, a backup interconnection tile is activated, while the faulty one is no longer used (Figure 3.3d), and it is reconfigured with an empty partial bitstream (i.e., a bitstream which does not contain any circuit information). The proposed architecture offers great flexibility as it does not rely on a fixed interconnection architecture among system modules (i.e., it can be applied to systems based on buses, point-to-point connections, interconnection networks, etc.). The idea of replicating the interconnection tile introduces negligible are overhead (as reported in Section 3.4). Despite an increment of the number of

input/output connections required bu each logic tile, this usually does not introduce any area overhead since each FPGA slice provides several input/output ports (this number depends on the FPGA device family). In addition, interconnection tiles do not require any computational element.

To implement our methodology the following number of configuration files, n_conf_files are required:

$$\begin{aligned} n\_conf\_files = n\_logic\_tiles \times n\_faults + \\ + n\_logic\_tiles^{n\_faults} \times (n\_faults + 1) + \\ + n\_logic\_tiles \end{aligned} \tag{3.1}$$

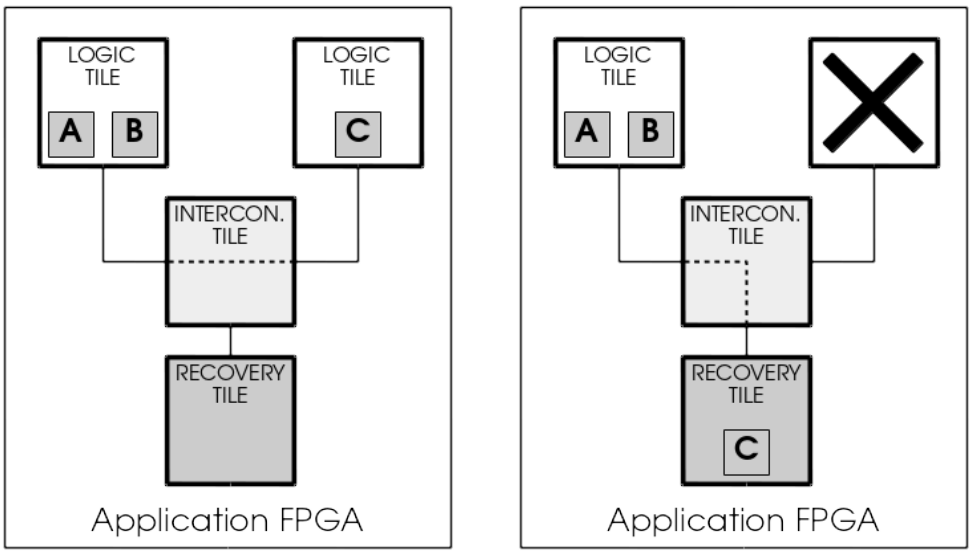
where n_logic_tiles is the number of logic tiles and n_faults is the number of permanent faults the system can tolerate. The first contribution is due to relocation of functions implemented in logic tiles to recovery tiles. The second contribution is due to interconnection tiles. In fact there are $n\_faults + 1$ interconnection tiles that must provide connection for all the possible combinations of logic tiles relocated to any of the recovery tiles. Finally, the third term of Equation 3.1 is referred to empty configuration of logic tiles needed after a relocation to a spare reconfiguration tile.

The proposed recovery methodology provides two main improvements with respect to [47]. The first one concerns the recovery time. In fact, it is reduced thanks to DPR enabling reconfiguration for just the faulty tile instead of the entire FPGA. Secondly, the memory required to store configuration files is dramatically decreased, as in [47] for each combination of sequences of faults a full bitstream was required.

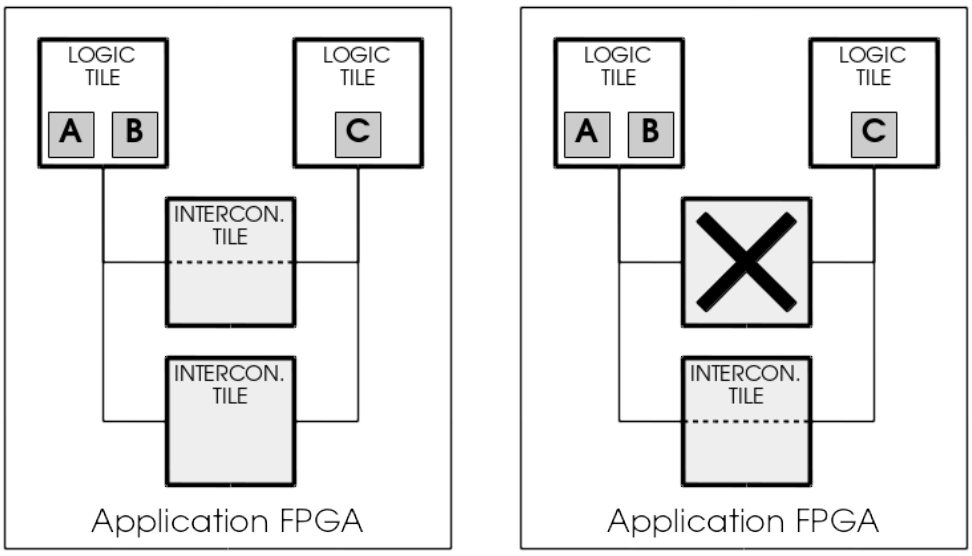
For the presented recovery strategy, the way the system is partitioned is extremely important as it influences the maximum number of permanent faults the system can tolerate. In fact, the largest number of recovery tiles that can be accommodated into the application FPGA depends directly on how basic components are grouped and distributed among logic tiles. The next subsection details a strategy for the partitioning of basic components maximizing the number of tolerable faulty tiles.

3.3.3 The proposed partitioning algorithm

Recovery tiles contain the necessary resources to accommodate every logic tile and their size is proportional to the number of slices, BRAM and DSP they offer. A



(a) The system without any faulty logic tiles (b) The system recovered from a faulty logic tile



(c) The system without any faulty interconnection tiles (d) The system recovered from a faulty interconnection tile

Fig. 3.3 The recovery strategies when a permanent faults occur. Fig 3.3a illustrates the relocation of the faulty logic tile into a recovery tile and the reconfiguration of the interconnection tile. Fig 3.3d shows how interconnections are reconfigured when a permanent fault is detected inside the active interconnection tile (Source: [1]).

fine-grained partitioning approach (e.g., assigning a basic component to each logic tile) leads to smaller recovery tiles, while a coarse-grained partitioning (i.e., grouping more than one component in a logic tile) requires larger tiles. In addition, changing partitioning means changing the number of wires inside the interconnection tiles, and so their area. Because of the relevance of the partitioning strategy, we propose an algorithm to find a feasible partitioning offering the maximum number of faulty tiles the system can tolerate (see Algorithm 1).

Algorithm 1 Partitioning algorithm.

```
Const N_components                          ▷ # of basic components
Const FPGA_res                              ▷ resources of the application FPGA
max_tolerated_faults = 0;
for n_logic_tiles= 1 to N_components do
    for each possible partitioning composed of n_logic_tiles do
        Slack_res = FPGA_res − Logic_tiles_res − Interc_tile_res
        n_tolerated_faults = Slack_res / (Rec_tiles_res+Interc_tile_res)
        if n_tolerated_faults > max_tolerated_faults then
            max_tolerated_faults = n_tolerated_faults;
            update best partitioning;
        end if
    end for
end for
```

As explained in the previous subsections, a model of the circuit is obtained by characterizing all the logic tiles with a number of resources and connections they require. The resources demanded by each logic tile depend on the basic component circuits it accommodates, while the number of connections is related to the partitioning. It is important to notice that interconnection tiles do not require any resource since they do not perform computation, instead they just need connections. As a consequence, the delay introduced by interconnection tile is assumed negligible for most of the applications (as will be shown in the next section) since the critical path is bounded by the logic circuits of basic components.

In Algorithm 1, *Rec_tiles_res* represents the resources needed by a single recovery tile, *Interc_tile_res* is an amount of FPGA slices required by a single interconnection tile, containing interconnections among all the logic and recovery tiles, while *Slack_res* are the spare resources when only logic tiles and one interconnection tile are taken into account.

Starting from the resources available on the target FPGA and the resources required by every basic component, the algorithm finds the best partitioning solution in terms of tolerable faulty tiles. In this case, $n\_tolerated\_faults$ is assumed in the worst case scenario, that is when permanent faults occur always in the same type of tile (i.e., logic/recovery tiles and interconnection tiles).

Basic components of a circuit are partitioned with different granularities. The number of logic tiles ranges from one, a single huge partition which represents the coarsest granularity, to the number of basic components defined in the modular design, the finest granularity. For each iteration all possible groups of basic component combinations are analyzed, and the number of tolerated faulty tiles is computed. To compute the number of tolerated faulty tiles for a given partitioning it is supposed that for each recoverable permanent fault there is one recovery tile and one backup interconnection tile.

The resulting number of tolerated faults, $n\_tolerated\_faults$, is the lowest among the ones computed for every type of resource (i.e., slices, BRAM, DPSs, ...). When $n\_tolerated\_faults$ is greater than the temporary maximum number of tolerated faults, $max\_tolerated\_faults$, it is saved as the best partitioning.

The proposed algorithm is complex from a computational point of view, that is, it scales badly with the increasing number of n_logic_tiles. However, since it must be executed just once at design-time, its execution time does not influence the overall performance of the implemented system. Building a fast partitioning algorithm is out of the scope of this paper. However, if a more efficient algorithm is needed, the proposed partitioning algorithm can be used as a starting point to model a Mixed Integer Linear Programming problem, as proposed in [47] or to design a genetic algorithm.

3.4 Experimental results

The proposed methodology was applied to two case studies in order to analyze and measure its performance: the first is *FEMIP* [65], an IP-core for images feature extraction and matching, and the second is the H.246 video encoder. The basic components of both the two systems were characterized by their resource require-

ments. Then, these values were fed into the proposed partitioning algorithm. The partitioning that maximizes the number of tolerated faulty tiles was chosen for the final system implementation. Performance are analyzed in terms of number of tolerated permanent faults, system recovery time and bitstream size. For both case studies, a Xilinx® *Virtex-4 VSX55* FPGA was chosen as the target *Application FPGA* since it provides a large number of slices, BRAMs and DSPs [66]. This FPGA can be dynamically and partially reconfigured by means of the *SelectMAP* port at a maximum reconfiguration throughput equal to 400MB/s [41].

As presented in Figure 3.4, *FEMIP* is composed of five basic components. FEMIP is characterized by a pipelined architecture. The number of interconnections between the pipeline stages is shown in Figure 3.4, while resources required by each component are reported in Table 3.1.

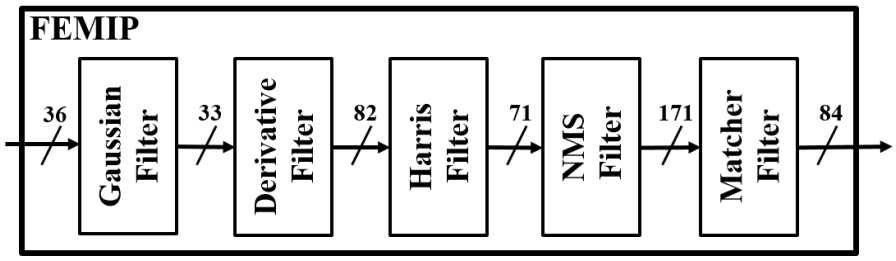


Fig. 3.4 FEMIP basic components (Source: [1]).

Table 3.1 Resources requirements for *FEMIP* basic components.

Component	# slices	# BRAMs
Gaussian filter	2560	8
Derivative filter	1344	8
Harris filter	3456	0
NMS filter	360	4
Matcher filter	650	6

Applying the proposed partitioning algorithm, the 5 basic components of *FEMIP* are organized into 4 logic tiles, as shown in Table 3.2. The *Application FPGA* contains all the logic tiles, 4 recovery tiles and 5 interconnection tiles. As a result, the implemented system is capable of tolerating up to 4 permanent faults. The number of partial bitstreams to store in the Bitstream Memory is equal to 1300

Table 3.2 Resources requirements for FEMIP tiles.

Tile	Components	# slices	# BRAMs	Bitstream
Logic 1	Gaussian filter	2560	8	254.5 KB
Logic 2	Derivative filter	1344	8	144.6 KB
Logic 3	Harris filter	3456	0	360.1 KB
Logic 4	NMS filter Matcher filter	1010	10	120.0 KB
Interconnection (x5)		128	-	7.2 KB
Recovery (x4)		3456	10	360.1 KB

and it is computed according to Equation 3.1. Even if this number is quite large, the total memory required is only 15.86MB. In detail, the amount to store the 16 bitstreams for the recovery tiles is 5.76MB. As the size of a single bitstream for an interconnection tile is as low as 7.2KB, the the amount of memory reserved for the bitstreams of interconnection tiles is 9.22MB, despite 1280 configuration files are generated. Finally, 879.2KB are employed for the storage of the bitstreams of the 4 empty logic tiles. As reconfiguration throughput is 400MB/s, the worst case recovery time in presence of a permanent fault is equal to 1.82ms (including the reconfiguration of the faulty logic tile with the empty bitstream, the configuration of the recovery tile and the update of the interconnection tile).

Thanks to this case study, it is possible to measure the performance overhead produced by interconnection tiles. The maximum operating frequency of *FEMIP* remains almost equal to the one obtained without the proposed fault tolerance technique (i.e., 60MHz). As a consequence, the introduction of the interconnection tile to allow communications between tiles does not affect performance of the system. This is due because of two reasons. First, the critical path is bounded inside the basic components logic. Secondly, the delay introduced by the interconnection tiles is 0.2ns, which is negligible with respect to the minimum clock period of *FEMIP* and of most of the applications usually implemented on FPGAs.

To compare the proposed methodology with respect to the one presented in [47], which also employs the Xilinx® *Virtex-4 VSX55* as *Application FPGA*, the H.246 video encoder was implemented. The comparison is performed in terms of number

of tolerated permanent faults, bitstreams size, and recovery time. Details about the 15 basic components of the H.246 video encoder and relative resources requirements can be found in [47]. Information about the partitioning selected by our algorithm is reported in Table 3.3. With the given partitioning, basic components are grouped

Table 3.3 Resources requirements for H.264 video encoder tiles.

Tile	**Components**	**# slices**	**# BRAMs**	**# DSPs**	**Bitstream**
Logic 1	intra8x8cc coretransform intra4x4 recon	3225	6	0	349.3 KB
Logic 2	dctransform calvc	2811	0	0	296.5 KB
Logic 3	buffer process1 quantise invdctransform dequantise invtransform	3123	6	6	380.3 KB
Logic 4	process2	3120	0	0	349.3 KB
Logic 5	header tobyte	1605	0	0	176.5 KB
Interconnection (x3)		320	0	0	18.0 KB
Recovery (x2)		3225	6	6	380.3 KB

into 5 logic tiles and it is possible to recover from 2 permanent faults, as in [47]. For this purpose 2 recovery tiles and 2 backup interconnection tiles were accommodated in the *Application FPGA*, in addition to the logic tiles and an interconnection tile. For our implementation the total amount of required *Bitstream memory* is 6605.2KB (90 configuration files in total), 3803KB for the recovery tiles (10 configuration files in total), 1350KB for the interconnection tiles (75 configuration files in total) and 1452.2KB for empty logic tiles (5 configuration files in total). In [47] 241 bitstreams are required for this case study. Although the number of bitstream that are obtained

applying the proposed methodology is almost 2.5x lower, the memory space required for bitstreams is dramatically reduced by 33x, from 291MB to 6.6MB. This reduction is enabled by DPR. In detail, the configuration of the entire FPGA is necessary to recover from a permanent fault in [47], thus resulting in a large size of each bitstream. Conversely, the proposed methodology adopts partial bitstreams, reducing the size of the configuration files.

Finally, the proposed methodology allows to shorten the recovery time since just a part of the implemented circuit reconfigured. In fact, in the worst case, 1.95ms are required to recover from a permanent fault, leading to a 4x improvement with respect to the 7.5ms [47] needed when a full reconfiguration of the FPGA is performed.

3.5 Conclusion

This chapter presents a novel methodology to increase availability and lifetime of FPGA-based systems affected by permanent faults. The methodology exploits Dynamic Partial Reconfiguration (DPR) to relocate faulty modules at run-time. A partitioning method is also presented to provide a solution which maximizes the number of permanent faulty modules the system can tolerate.
Experimental results highlight the negligible performance degradation introduced by applying the proposed methodology and the improvements in terms of both fault recovery time and memory requirements with respect to state-of-the-art solutions.
Possible future works could focus on a framework applying the proposed methodology automatically, thus requiring minimal efforts during the design phase. Moreover, the effectiveness of the proposed methodology with different granularity of basic components can be explored to find the best trade-off between number of tolerated faults and basic components size. Finally, the optimization of the partitioning algorithm to reduce its complexity by means of heuristics and genetic algorithms could be investigated.

Chapter 4

GPGPU Reliability evaluation

This chapter is the first one to be focused on reliability evaluation techniques. More specifically, it treats reliability of GPUs in the context of General Purpose computing on Graphics Processing Units (GPGPU). GPGPU offers a remarkable speedup for data parallel workloads, leveraging GPUs computational power. However, differently from graphic computing, it requires highly reliable operation in most of application domains. Characterization of the reliability of GPGPU systems is therefore becoming a mandatory task. This reliability analysis targets the AMD Southern Islands GPU architecture. SIFI is the reliability framework proposed in this chapter, able to evaluate reliability and to help systems engineers in the exploration of the design space, thus enabling system optimization. SIFI is based on Multi2Sim [67], a microarchitectural simulator. SIFI implements some of the most common reliability evaluation techniques for CPUs which are discussed and adapted to GPUs: Fault Injection (FI) and Architectural Correct Execution (ACE) analysis. SIFI evaluates reliability in presence of soft errors in the main memory arrays of the GPU (i.e., the vector register file, the scalar register file and the local memory). Since the main characteristic of GPGPU is performance, in this chapter a new metric combining reliability with performance is also presented, the Execution Per Failure (EPF). A comparison in terms of reliability between AMD Southern Islands and other NVIDIA GPU architectures is reported here to give an idea of the capability of this tool. This comparison was developed in collaboration with Sotiris Teselonis from University of Athens, who took care of the experiments about NVIDIA GPUs. The collaboration was funded by HiPEAC Collaboration Grant. Part of this chapter was previously published in [2, 68].

4.1 Introduction

Recent years have witnessed an increase of computational power demand in several application domains. GPGPU has gained a primary role in the delivery of high computational power leveraging the inherent high parallel architecture of GPUs to accelerate complex tasks. In this scenario, GPUs are no longer employed just for graphics, but they have increasingly found application in areas where reliability is a primary concern (i.e., advanced driver assistance systems, aviation, medicine, super computing, etc.). This trend is however threatened by the technology shrinking, which has a detrimental effect on the susceptibility to faults for new devices (especially for large storage arrays) [69]. Characterization of the reliability of GPGPU systems is therefore becoming a mandatory task.

One of the main opened challenges in evaluating the reliability of GPGPU systems is the development of fast and accurate reliability assessment tools, able to properly trade-off simulation time and accuracy. Some benefits would be introduced by providing information to guide the system designers in the choice of proper architectural parameters and error protection mechanisms to achieve the target reliability and performance requirements. Recently, Tselonis and Gizopoulos proposed a reliability analysis framework based on microarchitectural level fault injection able to analyze the effect of soft errors in systems based on NVIDIA GPU chips [70]. This framework represents a valuable instrument for GPGPU system designers as reported in [71] where it was exploited for the reliability analysis of heterogeneous systems. Similar tools able to analyze systems based on AMD GPU chips that together with NVIDIA cover almost the totality of the GPU market share are still not mature and available. A reliability study using fault injection to analyze systems based on the old AMD Evergreen GPU architecture was presented in [72]. However, no results are reported in the literature for the analysis of the newer AMD Southern Islands architecture.

This chapter presents SIFI (Southern Islands Fault Injector), a framework for the reliability analysis of systems based on the AMD Southern Islands GPU architecture in presence of soft errors. Using SIFI, reliability can be assessed not only by means of fault injection but also using very fast ACE analysis [15]. In both cases, a SEU in the main memory arrays of the GPU (i.e., the vector register file, the scalar register file and the local memory) is the considered fault model. SIFI implements a set of techniques to reduce simulation time of fault injection campaigns, thus allowing the

analysis of complex systems executing realistic software applications. Moreover, SIFI offers the possibility to perform reliability analysis just considering the portion of the hardware resources actually used by the running software, thus decoupling the reliability assessment from the occupancy of the target architectural hardware components, and focusing on the analysis of the resiliency of the executed software to the injected faults. Since SIFI is built on top of the Multi2Sim [67] microarchitectural simulator, it can be easily extended to the architectures supported by this simulator, including the AMD Evergreen architecture. The optimized simulation environment makes SIFI a valuable tool to assist designers when taking decisions on specific GPU architectural parameters. Several system configurations can be analyzed and compared in order to identify the best configuration given the application constraints.

In order to show the potential of the proposed reliability framework, a reliability study is performed considering 14 GPGPU applications executed on different AMD Southern Islands GPUs.

In addition, GPUs from different vendors, architectures and programming models are compared: AMD Southern Islands, NVIDIA G80, GT200 and Fermi. For this purpose, while AMD GPU architecture is evaluated by SIFI, reliability of NVIDIA GPU architectures is assessed by GUFI [70], a tool similar to SIFI, but for NVIDIA GPUs. In this case, reliability of all devices is analyzed running the same set of 10 benchmarks, written using the OpenCL language [15] for the AMD GPU and the CUDA language [16] for the NVIDIA GPUs. The comparison is performed employing both fault injection campaigns and ACE analysis. The use of microarchitecture level simulators provides significant flexibility and leads to a dramatic simulation time reduction compared to the RTL models, which are often not publicly available. Such a multidimensional study delivers significant insights on: differences in GPU vulnerability estimations between fault injection experiments and ACE analysis; variations in the vulnerability of specific hardware components and benchmarks among different GPU architectures; joint evaluation of reliability and performance to support designers and programmers when evaluating different GPUs and workloads.

4.2 Related works

GPUs reliability literature includes publications that focus either on reliability evaluation only or present and evaluate GPUs fault tolerance solutions for both soft- and hard- errors[72–83].

Previous works on GPUs reliability evaluation usually involve the use of microarchitectural simulators for the AVF estimation of selected hardware structures. For these estimations, either fault injection or ACE analysis is considered separately [75, 76, 81, 70]. In the presented methodology, both methods in both AMD and NVIDIA GPUs were used. ACE analysis has already been applied on top of the publicly available GPGPU simulators GPGPU-sim [84] and Multi2sim [67]respectively. However, ACE analysis for CPUs is known to overestimate the vulnerability of the hardware structures [85, 86]. Using both Fault Injection and ACE analysis for the evaluation of the vulnerability of the hardware components, as for this work, insights into this overestimation can be shown. In the reported evaluation, this overestimation is measured in the register file and local/shared memory; there are several cases of benchmark and GPU chip combinations where the overestimation is large and others where ACE reports virtually the same AVF as fault injection.

Fault Injection has been used for GPGPUs in [83] and [73] but not at the microarchitecture level as for this work. Both [83] and [73] inject faults only on architectural registers that are accessed during the execution of randomly selected instructions (fault injection at the software level) while the proposed framework enable us to model accurately the effect of a fault which strikes on a hardware component at a randomly selected execution cycle (fault injection at the microarchitectural level).

The works of [72, 82, 70] employ microarchitectural simulators to inject faults. Both [72, 82] use Multi2Sim, to inject faults in the components of an AMD Evergreen GPU running OpenCL workloads while we inject faults into the hardware components of a newer AMD Southern Island GPU architecture. In [70], GPGPU-Sim is used to inject faults in the hardware components of a Fermi architecture while in the presented evaluation more NVIDIA GPU architectures are analyzed, using both fault injection and ACE analysis on GPGPU-Sim.

4.3 The Southern Islands AMD architecture

From the hardware standpoint, the Southern Islands AMD architecture consists of several compute units (CUs) sharing a global memory and managed by a scheduler (Figure 4.1). Each CU is composed of a front-end that fetches instructions and dispatches them to the appropriate unit: a branch unit, a local data storage unit performing operations on local memory, a scalar unit executing scalar operations on scalar registers and several SIMD units containing a vector ALU,the integer and floating point units operating on the vector register file in parallel. OpenCL is the programming model used by the Southern Islands AMD architecture [87].

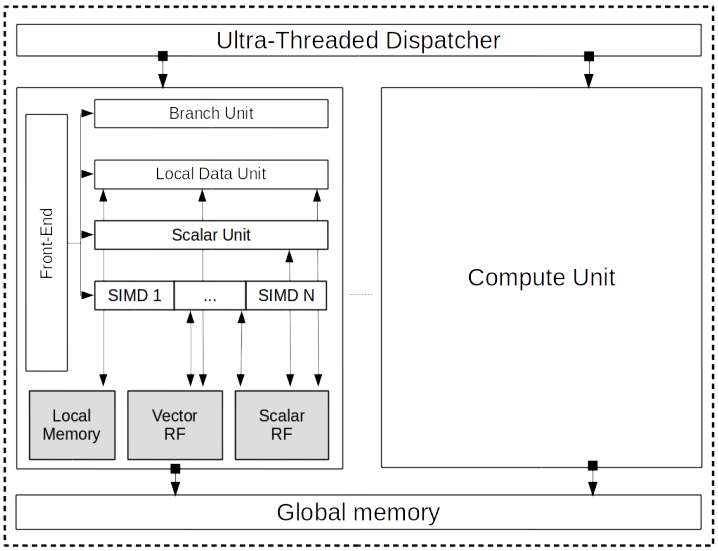


Fig. 4.1 Southern Islands AMD architecture.

In OpenCL, parallel portions of an application (kernels) are executed on the GPU as work-items. Work-items are grouped into work-groups. Communication among work-items is possible only for work-items belonging to the same work-group by means of local memory. When a new kernel is executed a new ND-Range is created. The programmer specifies the number of work-groups and the number of work-items per work-group characterizing the ND-Range.

Every time the GPU can execute a new work-group, the work-group is assigned to a CU. The number of work-groups a CU can concurrently execute ($\#wg$) depends on the maximum number of work-groups a CU can accommodate ($\#maxwg$) and on the amount of resources required by a single work-group. More specifically, each

work-group requires a certain amount of resources ($\#v_{wg}$ vector registers, $\#s_{wg}$ scalar registers and $\#lm_{wg}$ local memory cells) out of the total resources available in the CU ($\#V_{RF}$ vector registers, $\#S_{RF}$ scalar registers and $\#LM$ local memory cells). The number of concurrent work-groups, $\#wg$, is chosen according to [88]:

$$\#wg = min(\#wg_v, \#wg_s, \#wg_{lm}, \#max\ wg) \tag{4.1}$$

where:

$$\#wg_v = \left\lfloor \frac{\#V_{RF}}{\#v_{wg}} \right\rfloor, \quad \#wg_s = \left\lfloor \frac{\#S_{RF}}{\#s_{wg}} \right\rfloor,$$
$$\#wg_{lm} = \left\lfloor \frac{\#LM_{RF}}{\#lm_{wg}} \right\rfloor$$

Finally, in order to be executed in parallel, work-items of the same work-group are grouped into wavefronts. The number of work-items per wavefront depends on the parallelism of the SIMD Units of the chip.

4.4 SIFI architecture and functionalities

SIFI is built on top of Multi2Sim [67] v. 4.2, a microarchitectural simulator for heterogeneous systems modeling accurately the microarchitecture of the AMD Southern Islands GPUs. The analysis carried out by SIFI focuses on soft errors in the main memory arrays of the GPU (i.e., the vector register file, the scalar register file and the local memory). These faults are relevant for large memory arrays such as the one considered by SIFI. Analysis of multiple bit upsets is not considered, however it can be easily implemented.

SIFI measures the reliability of a GPU based system by computing the Architectural Vulnerability Factor of its hardware structures [15]. The AVF quantifies the probability of a soft error to manifest as a failure of the system jointly considering the masking properties of the hardware architecture as well as of the executed software. Moreover, since the AVF is strongly influenced by the actual occupancy of the hardware structures, to allow a deep understanding of the contribution of the instruction flow on the error masking, SIFI also enables to compute the AVF Util metric introduce by Farazmand et al. in [72]. The AVF Util is the probability that a

soft error in a *used* hardware structure causes a system failure. The relation between AVF and AVF Util can be expressed as:

$$AVF = AVF_{Util} \times Occupancy \tag{4.2}$$

Once the AVF is estimated for each GPU hardware structure, the Failure In Time (FIT) rate of the system (λ_S) can be computed combining size and vulnerability of every hardware structure of the GPU:

$$\lambda_S = \sum_{i \in \{vRF, sRF, LM\}} AVF_i \times \lambda \times \#bit_i \tag{4.3}$$

where $\#bit_i$ is the number of memory elements of the hardware structure i and λ is the raw error rate per bit of the target technology node.

Since the FIT rate is a pure reliability metric and does not provide any information about the system performance, SIFI enables the computation of a new reliability metric named Executions Per Failure (EPF). EPF is the number of times an application must be executed before observing a system failure. It is computed as

$$EPF = EIT / \lambda_S \tag{4.4}$$

where EIT (Executions in Time) is the number of executions of an application in 10^9 hours of device operation. The EPF enables to jointly analyze performance and reliability into a single metric.

Another metric that can be defined to jointly evaluate reliability and performance is the Instructions Per Failure (IPF). The IPF measures the instruction throughput of a benchmark between failures instead of the complete program execution time like in the EPF:

$$IPF = IIT / \lambda_S \tag{4.5}$$

where IIT is the Instructions In Time, the number of instructions executed in a billion of hours of computation. EPF and IPF are relateted each other as:

$$IPF = EPF \times \#inst \tag{4.6}$$

where $\#instr$ is the number of instructions executed by the application.

SIFI can compute the presented metrics using different simulation engines described in the following subsections.

4.4.1 Fault injection engine

The fault injection (FI) engine is the most accurate simulation engine available in SIFI. It performs reliability analysis by simulating the occurrence of faults in the GPU hardware structures considering a statistically significant number of program executions (one fault per execution) [89]. The impact of a fault on the system is evaluated by comparing the output of the computation with the one of a golden execution. At a high level, the impact of the fault is classified as masked or non-masked[1]. Since FI is a computational intensive task, SIFI is designed to speedup the FI campaign trying to reduce the required number of simulations without loosing accuracy. A FI campaign consists of several steps. At first the application is profiled in order to identify the time intervals in which the GPU is active and to gather information about the executed kernels. The faults to be injected are then randomly generated and another simulation is performed to profile whether these faults strike a hardware structure actually assigned to one of the work-groups. In case a fault hits a non-assigned hardware structure, it is marked as masked without performing any simulation. Otherwise, it is marked as *Util*. Eventually, all faults marked as *Util* are simulated and classified. Using the results of FI simulations the AVF and AVF Util of an hardware structure can be computed as:

$$AVF = \frac{\#\,inj_{not-masked}}{\#\,inj} \qquad AVF_{Util} = \frac{\#\,util\ inj_{not-masked}}{\#\,util\ inj} \tag{4.7}$$

The speedup obtained by skipping *non-Util* simulations depends on the application and mainly on the occupancy of the hardware structures and can be computed as:

$$S = \#\,inj/\#\,util\ inj \tag{4.8}$$

[1] Fine grained classification of non-masked faults into Silent Data Corruption (SDC) and Detectable Unrecoverable Error (DUE) is also possible

4.4.2 ACE analysis engine

Unlike FI, the ACE analysis engine requires just a single simulation of the application to perform AVF estimations. It is therefore a very fast analysis that however has reduced accuracy with respect to FI. SIFI ACE analysis engine is based on the techniques presented in [15] and [90] for CPU memory arrays.

Let us consider the computation of the AVF for the vector register file (a similar procedure can be applied to the other hardware structures). The ACE analysis is based on the principle that not all registers in this register file are continuously involved in the computation and the AVF of the hardware structure can be estimated by determining which registers affect the final system output (ACE registers) and which do not (un-ACE registers).

When performing ACE analysis each kernel is analyzed separately and then results are recombined together. For each kernel, the amount of registers assigned to each generated work-group ($\#v_{wg}$) is computed (see Section 4.3). All registers not assigned to any work-group are classified as *idle* and directly marked as un-ACE, while the others are profiled during the execution of the kernel. During the time intervals (i.e., clock cycles) between a read and a write operation (*read-to-write* intervals), and between two consecutive write operations (*write-to-write* intervals) the register can be safely considered un-ACE. In all other cases it is marked as ACE.

To reduce complexity and to implement a very fast reliability analysis workflow, SIFI does not consider more sophisticated techniques, which also take into account dead data (i.e., data not contributing to the output results) and logic masking (i.e., logical and arithmetic operations resulting in masked results). Their computation significantly increases the complexity of the analysis and therefore has not been implemented in this study where ACE analysis is mainly exploited for its fast simulation time. It is however worth to report that, from our simulations, we noticed that dead data in the selected benchmarks represent a negligible portion of the application (less than 0.5%) and, therefore, neglecting them does not introduce a significant loss of accuracy.

The ACE factor of each work-group (i.e., the work-group average number of ACE registers per clock cycle) can therefore be computed as:

$$ACE_{wg} = \sum_{i}^{\#v_{wg}} \frac{ACE_{clk-vreg-i}}{wg_{clk}} \tag{4.9}$$

where wg_{clk} is the number of clock cycles required to execute the work-group and $ACE_{clk-vreg-i}$ is the number of clock cycles in which the register i is classified as ACE. The ACE factors of each work-group can be combined together to compute the ACE factor of the compute units (ACE_{CU}). To perform this computation, a timing diagram representing the time window of every work-group executed by the CUs is built. Figure 4.2 is an example of timing diagram for a single CU assigned to 4 work-groups and able to execute 2 work-groups concurrently.

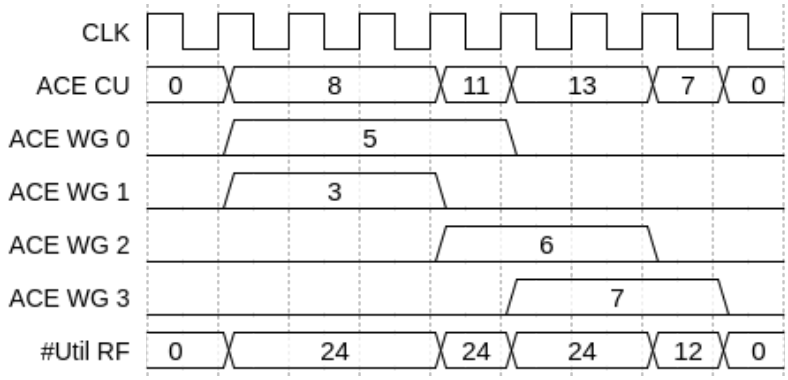


Fig. 4.2 An example of ACE_{CU} timing diagram for a single CU. Considering $\#V_{RF} = 32$ and $\#kclk = 7$, then $AVF = \frac{\frac{8+8+8+11+13+13+7}{32}}{7} = 0.3$. If $\#v_{wg} = 12$ then $AVF_{Util} = \frac{\frac{8}{24}+\frac{8}{24}+\frac{8}{24}+\frac{11}{24}+\frac{13}{24}+\frac{13}{24}+\frac{7}{12}}{7} = 0.45$

The ACE factor of the CU at clock cycle i ($ACE_{CU}(i)$) is computed by summing the ACE_{wg} of its active work-groups on that clock cycle (Figure 4.2). Finally, the AVF of the entire vector register file is computed as:

$$AVF = \frac{\sum_{j}^{\#CU} \sum_{i}^{\#kclk} \frac{ACE_{CU_j}(j)}{\#V_{RF}}}{\#kclk} \tag{4.10}$$

where $\#kclk$ is the number of clock cycles required to execute the GPU kernel, $\#CU$ is the number of available compute units and $\#V_{RF}$ is the number of registers per compute unit. The computation of the AVF takes into account the ratio between the number of ACE registers and the total number of registers available in the vector

register files of the GPU. Similarly to the AVF, by considering the average number of used vector registers of the active work-groups of each CU ($\#util\ RF_j$ in Figure 4.2) instead of the total available registers $\#V_{RF}$, the AVF Util can be computed as:

$$AVF_{Util} = \frac{\sum_{j}^{\#CU} \sum_{i}^{\#kclk} \frac{ACE_{CU_j}(j)}{\#util_{RF_j}(i)}}{\#kclk} \tag{4.11}$$

4.5 Experimental results

Results related to SIFI and a comparison between AMD and NVIDIA GPUs are presented in this section. At first, SIFI results are reported to demonstrate the advantages introduced by this framework. Secondly, NVIDIA and AMD GPUs are compared under several aspects related to reliability and performance.

4.5.1 SIFI results

This subsection aims at demonstrating the capability of SIFI when analyzing the reliability of a set of benchmark systems.

Experimental Setup

For our evaluation we consider 14 software benchmarks with SIFI configured to resemble the architecture of the AMD HD Radeon 7970 GPU device[2]. A CU of this GPU consists of 4 SIMD Units. The scalar register file is composed of 2K 32-bit registers while the local memory size is 56KB. The scalar register file and the local memory are shared among all the SIMD Units. Moreover, each SIMD Unit features a vector register file of 56K 32-bit registers. Starting from this basic configuration, to demonstrate the capability of SIFI when analyzing different GPU architectures, experiments considering CUs with different numbers of SIMD units are performed as well. The data workload of each benchmark is chosen to maximize and stress the use of the CU memory arrays that are considered by the analysis. However, this significantly increases the simulation time when considering multiple CUs. To tackle

[2]Any chip belonging to the AMD Southern Islands family can be modeled

with this issue, following the approach proposed by Farazmand et al. in a similar GPU study [72], we scaled the analysis considering a single CU.

The benchmarks considered in our experiments are selected from AMD-APP-SDK benchmarks[3]: (1) Binary Search (BinS), (2) Bitonic Sort (BitS), (3) DCT, (4) DwtHaar1D (DWT), (5) FastWalshTransform (FWT), (6) FloydWarshall (FW), (7) Histogram (HIS), (8) MatrixMultiplication (MM), (9) MatrixTranspose (MT), (10) QuasiRandomSequence (QRS), (11) RecursiveGaussian (RG), (12) Reduction (RED), (13) SimpleConvolution (SC) and (14) URNG.

For each benchmark we exploited SIFI to compute the *AVF* and *AVF Util* of the target system resorting both to the FI and ACE analysis engines described in Section 4.4. According to [89], for each fault injection campaign (i.e., for each benchmark and for each hardware structure), we applied statistical fault sampling, injecting a number n of faults equal to:

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p))}} \tag{4.12}$$

where N is the population size[4], p is the estimated probability of a fault to generate a failure[5], e is the accepted error margin[6] and t is the cut-off point that defines the confidence level[7]. Considering the simulated benchmarks and the target GPU architecture, to characterize a hardware structure for a given benchmarks about 10K fault injections were performed.

Results

Figure 4.3 reports the AVF of the vector register computed using both FI and ACE analysis. Results show how different software applications can significantly influence the AVF thus confirming the need to carefully perform this type of analysis. The benchmark with highest vulnerability is MM (21%), while some benchmarks are characterized by very low AVF (i.e., BinS, BitS, DWT, FW, RG, RED and URNG).

[3] AMD-APP-SDK v.2.7 available at: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/

[4] the size of the targeted memory array multiplied by the number of clock cycles of the application

[5] since it is unknown, a typical value of 0.5 is used to maximize the sample size

[6] 1% in our case

[7] 95% in our case

As expected, ACE analysis provides a rough estimation of AVF. In most of the cases, it overestimates more than two times the vulnerability. This difference can be attributed to the fact that ACE analysis does not take into account dead instructions and software logic masking. Interestingly, when considering the AVF of the local memory results obtained using the ACE analysis are quite accurate (Figure4.4) and in general fall within the error margin of the estimations obtained using FI with the only exception of MT, that is 8 percentile points higher. HIS has the highest local memory vulnerability (91%). This is due to the fact that it requires a large amount of memory that is employed as a read-only buffer.

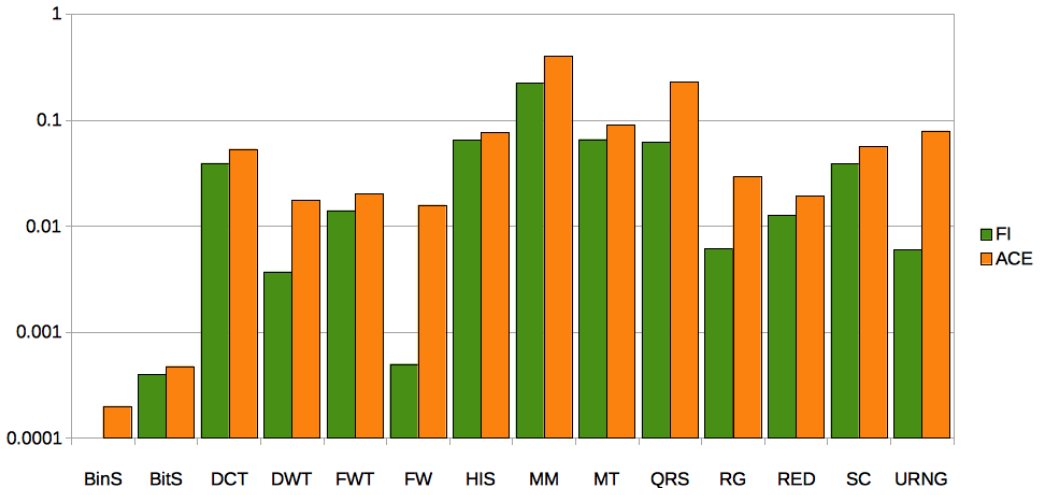


Fig. 4.3 Vector register file AVF computed by FI and ACE analysis.

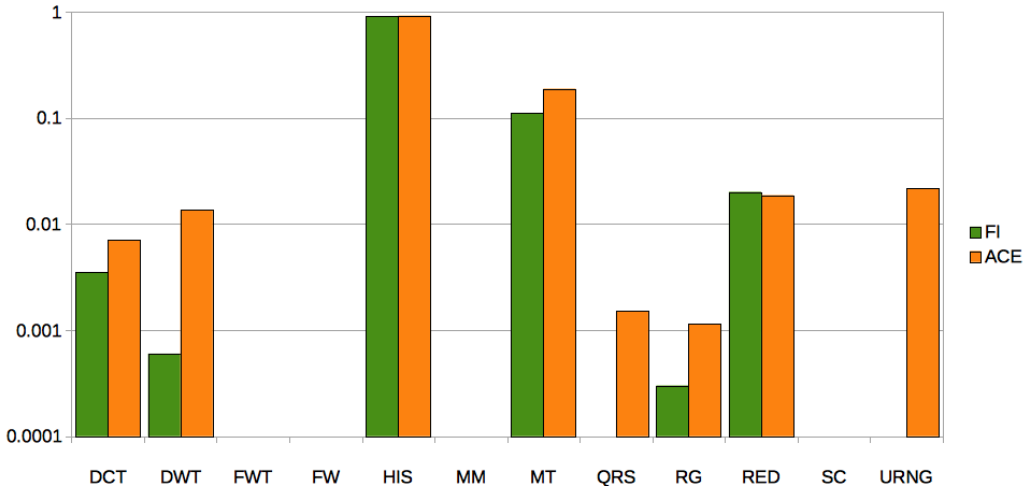


Fig. 4.4 Local memory AVF computed by FI and ACE analysis. The AVF is reported just for benchmarks using local memory.

Finally, Figure 4.5 reports the AVF for the scalar register file. It ranges between 0.2% (BinS) and 16% (MM) with an average value of 6.3%. ACE analysis estimation is close to the one obtained by FI just for DCT, while, in the other cases, the AVF is assessed as almost the double of FI.

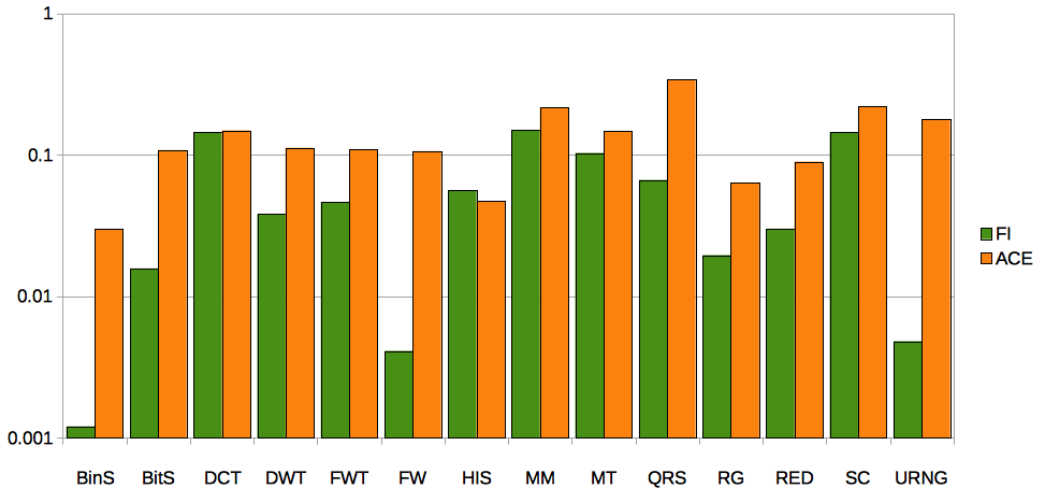


Fig. 4.5 Scalar register file AVF computed by FI and ACE analysis.

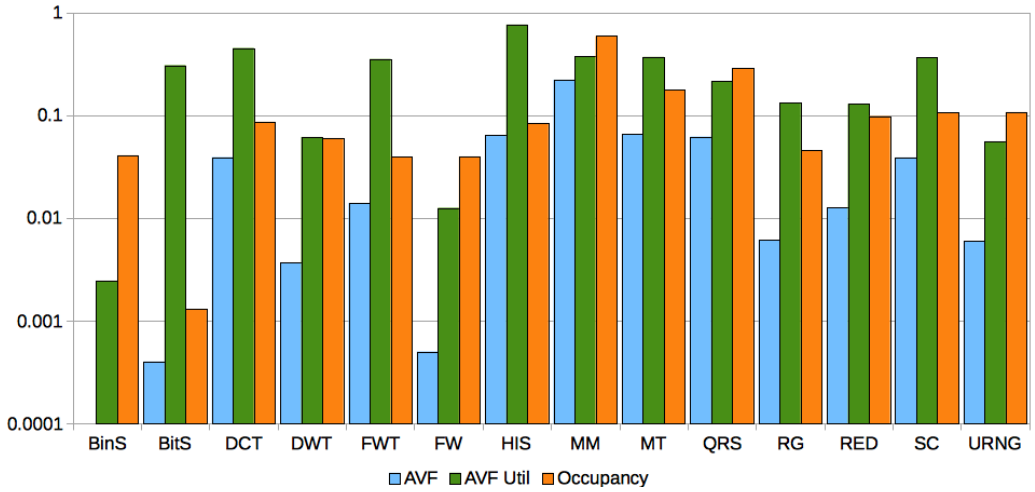


Fig. 4.6 The correlation between occupancy and AVF of the vector register file. The AVF Util is computed in order to decouple vulnerability and occupancy.

According to Section 4.4, the AVF is influenced by two factors: the occupancy of the memory arrays and the AVF Util. SIFI allows us to analyze these contributions separately as reported in Figure 4.6 for the vector register file. The results show clearly that occupancy is one of the most relevant contributions to the AVF. Let us

consider results for HIS. The AVF Util of this benchmark is equal to 80.1%, showing that this application is potentially highly vulnerable to faults striking active resources. However, this high vulnerability is compensated by a low occupancy of the register file that leads to a total AVF of only 6.7%. For benchmarks with higher occupancy (e.g., MM), the difference between AVF and AVF Util is reduced. Being able to analyze the relationship between AVF, AVF Util and Occupancy is an important instrument to carefully plan the introduction of fault tolerance mechanisms in the system when and where required. Results similar to the one reported in Figure 4.6 are also obtained for the local memory and for the scalar register file with similar trends, therefore are not reported.

One of the main benefits of SIFI is the possibility to evaluate the reliability of a system exploring different architectural parameters. To stress this capability we show how the number of SIMD units per CU affects the AVF of the vector register file (Figure4.7), local memory (Figure4.8) and scalar register file (Figure4.9). Changing the number of SIMD units leads to AVF variations. More specifically, decreasing the number of SIMD units increases the AVF of the vector register file, while it reduces the vulnerability of the local memory and the scalar register file. This is an interesting behavior that requires further investigation. However, some considerations can be drawn. Concerning the local memory and the scalar register file, AVF variations can be justified since increasing the number of SIMD units increments the required bandwidth and consequently the latency to main memory. This results in a larger time-window of vulnerability for a single memory element that leads to an increment of the AVF. On the opposite, the trend for the vector register file is unexpected.

To conclude the proposed reliability analysis, Figure 4.19 reports the FIT rate, the EIT and the EPF computed by SIFI for the analyzed systems. These metrics allow us to introduce the contribution of the technology and performance into the analysis. The FIT rate has been computed considering a raw failure rate per bit of λ = 1mFIT/bit.

From the performance prospective (EIT), decreasing the number of SIMD units always translates into longer execution time, however in some cases, as BinS, HIS and RED, this overhead is negligible. Concerning reliability, the FIT is improved in all the cases when the number of SIMD unit is reduced, apart from QRS which has an opposite trend. Finally, thanks to this analysis it is possible to combine both performance and reliability, taking into account the EPF, that is the expected value

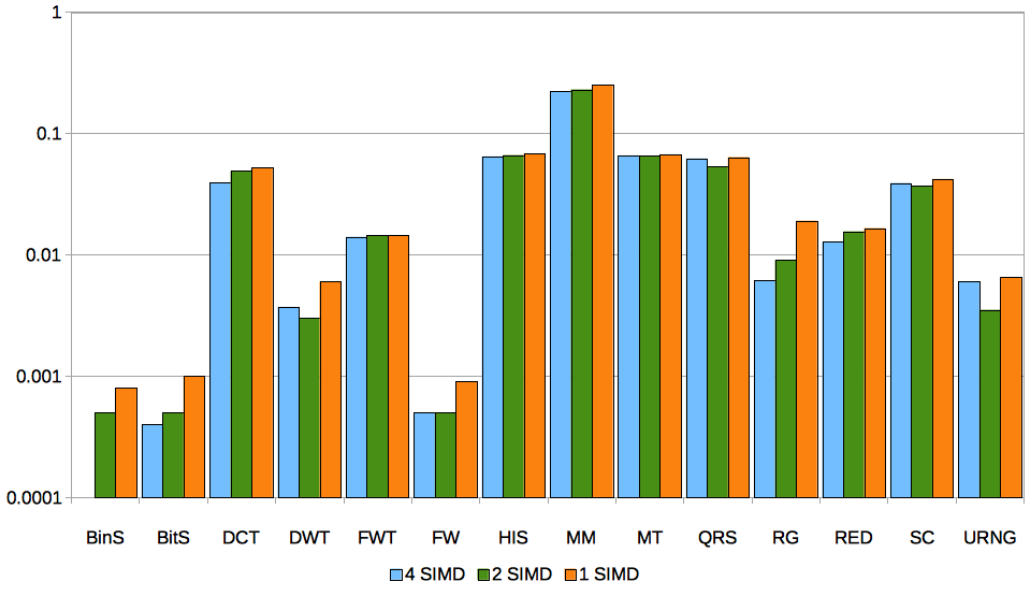


Fig. 4.7 Vulnerability comparison of vector register file changing the number of SIMD Units per CU.

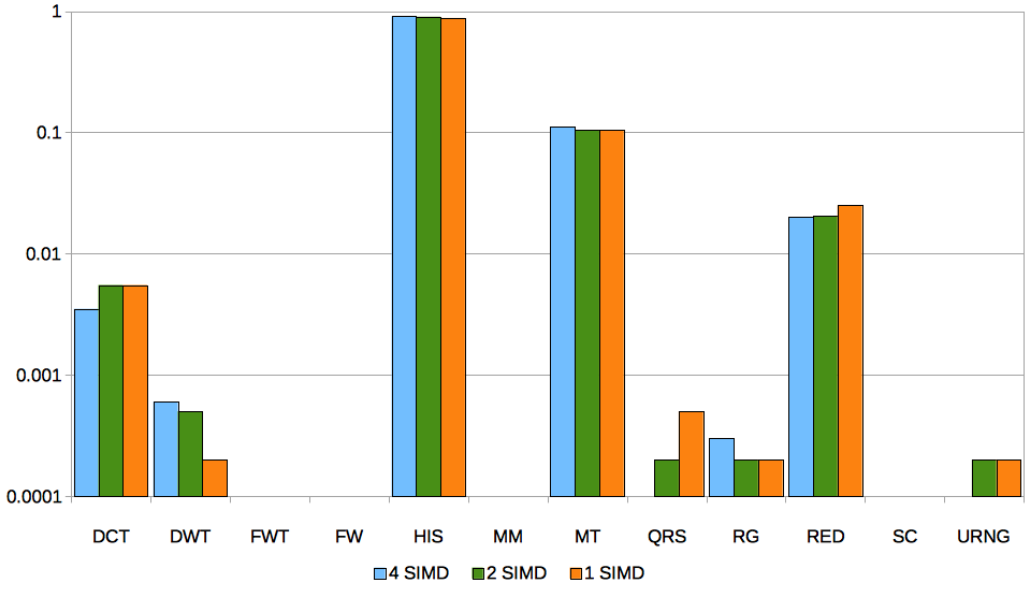


Fig. 4.8 Vulnerability comparison of local memory changing the number of SIMD Units per CU.

of application executions before a failure manifests. In this case, both the EIT and FIT decrease for smaller numbers of SIMD units. As a consequence, the parameter that decreases faster dominates. On average, the FIT prevails leading to higher EPF for a single SIMD unit and lower EPF in case of 4 SIMD units. However the trend is

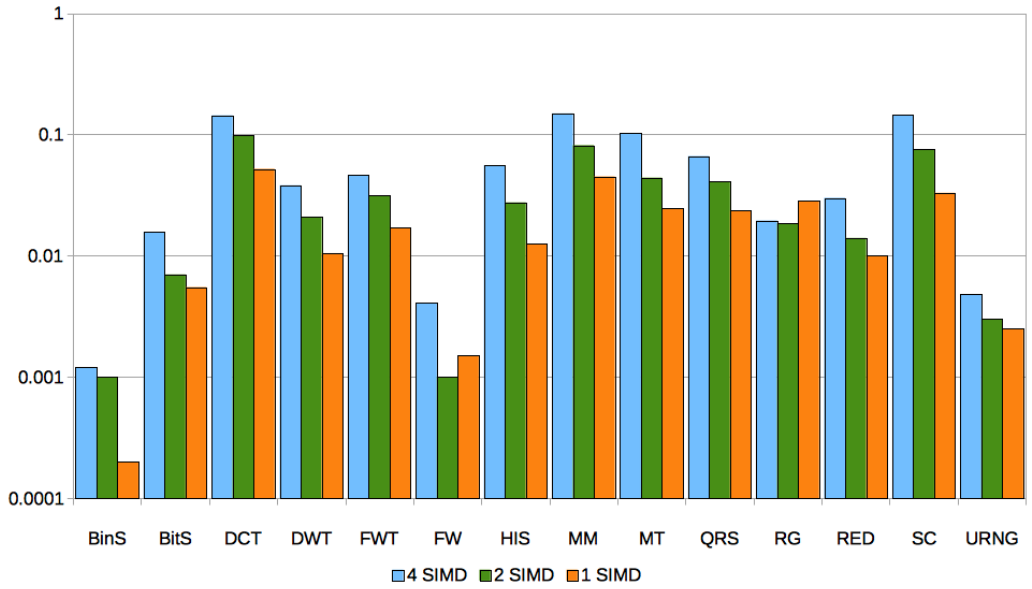


Fig. 4.9 Vulnerability comparison of scalar register file changing the number of SIMD Units per CU.

not the same for all the analyzed benchmarks. In fact, BitS, DCT and MM show the opposite behavior. This strongly suggests that EPF should be evaluated separately for each application not to commit errors in reliability assessment and during the design phases of a GPU-based system.

Finally, results illustrating the performance of SIFI as well as the capability of executing complex benchmarks in reasonable time are presented in Figure 4.10. In particular, the advantage introduced by the speedup technique of SIFI fault injector can be appreciated. In fact, the time required by a fault injection campaign is often reduced by almost one order of magnitude. In addition, for each benchmark, the number of simulated GPU instructions is reported too.

4.5.2 A multi-faceted comparison of reliability between AMD and NVIDIA GPU architectures

The results presented in this section compare AMD and NVIDIA GPU architectures on the basis of their reliability and performance. This work was developed in collaboration with University of Athens. In detail, they developed GUFI [70], a tool similar to SIFI but targeting NVIDIA GPUs. GUFI can perform both fault injection and ACE analysis for NVIDIA Quadro™FX 5600 (G80 architecture),

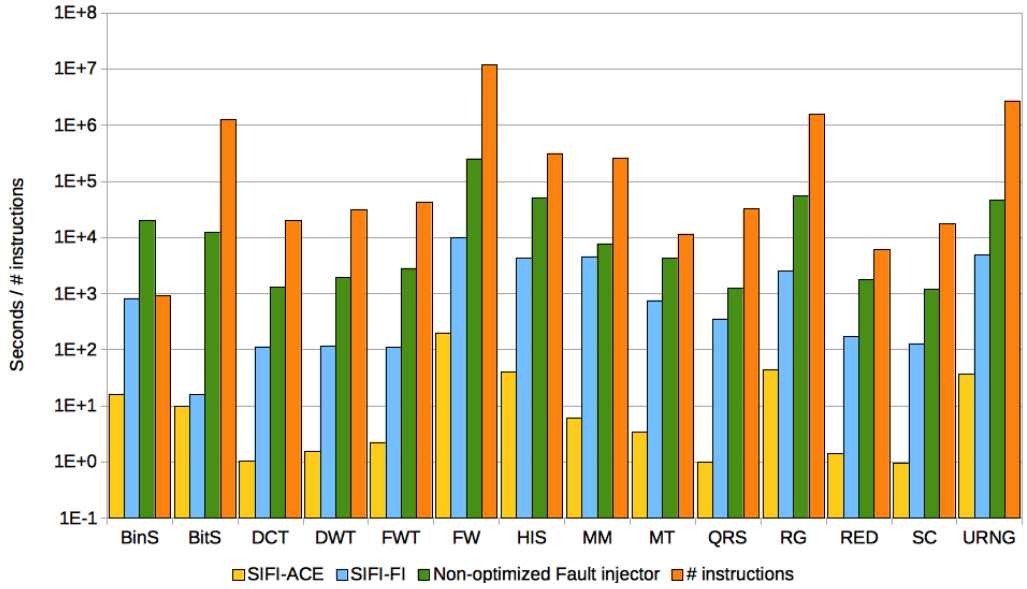


Fig. 4.10 SIFI timing performance. For each benchmark the figure reports the time required to estimate the vector register file AVF using ACE analysis, FI and non-optimized FI (10,000 injections using 8 cores) alongside the number of GPU instructions per simulation.

NVIDIA Quadro™FX5800 (GT200 architecture) and NVIDIA Geforce™GTX 480 (Fermi architecture). Moreover it is able to compute all the reliability metrics introduced in the previous sections, so that a fair comparison can be performed. For this experiments the AMD HD Radeon™7970 (Southern Islands architecture) was selected among AMD GPUs. The hardware structures considered for this analysis are the vector register file and the local memory (the general purpose register file and shared memory, using NVIDIA terminology). For the reminder of this chapter OpenCL/AMD terminology is employed even when referring to NVIDIA GPUs.

Experimental Setup

For our evaluation, we used 10 benchmarks: 7 available both in the CUDA SDK[8] and AMD-APP SDK[9] and 3 from Rodinia benchmarks [91]. For every benchmark both the CUDA implementation for the NVIDIA chips and the OpenCL implementation for the AMD chip is available.

[8]https://developer.nvidia.com/cuda-toolkit-42-archive

[9]http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/

Table 4.1 summarizes the main characteristics of the CUs of the GPU chips analyzed in this work.

Table 4.1 CU details of the target GPU architecture.

	Chip name	**Quadro™ FX5600**	**Quadro™ FX5800**	**Geforce™ GTX 480**	**HD Radeon™ 7970**
	Architecture	**G80**	**GT 200**	**Fermi**	**Southern Islands**
Register file		32KB	64KB	128KB	256KB
Local memory		16KB	16KB	48KB	64KB
SIMD Units		1	1	2	4
Max	#wg	8	8	8	40
	#wavefronts	24	32	48	40
	#work-items	768	1024	1536	1840

For both the AMD and the NVIDIA architectures, we measured the AVF of the vector register file and the local memory using both Fault Injection (FI) and ACE Analysis (ACE) as described in the previous section. For the FI experiments, we applied statistical fault sampling [89], thereby making 2,000 fault injection experiments for each combination of GPU architecture, and hardware component (vector register file, local memory) corresponds to 2.88% error margin with 99% confidence level. To perform a fair comparison, every benchmark was executed with the same input data set for all considered GPU devices. The data workload of each benchmark was chosen to maximize and stress the use of the CU memory arrays considered during the analysis. However, this significantly increased the simulation time when considering multiple CUs. To tackle with this issue, following again the approach proposed by Farazmand et al. in a similar GPU study [72], we scaled the analysis considering a single CU.

Results

We start the analysis of the reliability of the different GPU architectures and benchmarks by looking at the AVF summarized in Figure 4.11 for the register file and Figure 4.12 for the local memory. The two charts report the AVF computed using both FI and ACE analysis.

By analyzing the register file (Figure 4.11), we can observe significant changes in the AVF depending both on the hardware platform and on the executed benchmarks.

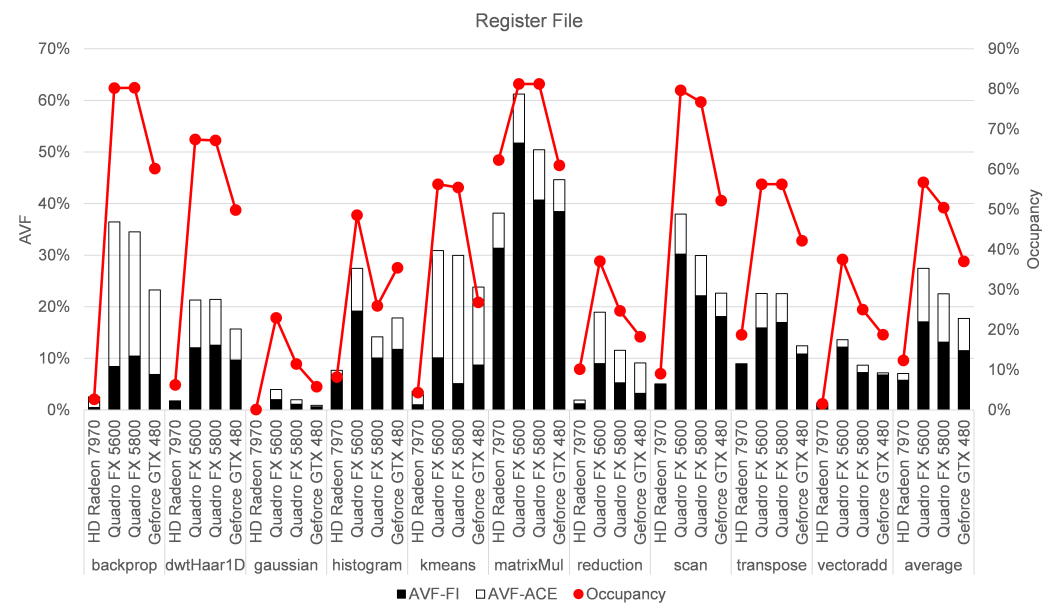


Fig. 4.11 AVF for Register File measured by FI and ACE analysis (Source: [2]).

Overall, the HD Radeon 7970 is the chip with lowest register file vulnerability while the Quadro FX 5600 is the one with highest vulnerability for all benchmarks. Looking at a single hardware platform, the difference in the AVF for the different benchmarks is the result of the way the software stresses the resource and is able to be resilient to the injected faults. Instead, when looking at the same benchmark executed on the different chips we interestingly note a strong correlation of the AVF with the register file occupancy (red bullets). Based on their average occupancy, the architectures can be ordered as follows: HD Radeon 7970 (12.89%), GTX480 (37%), Quadro FX5800 (50%), Quadro FX 5600 (57%). This trend is respected for all benchmarks with the exception of histogram in which the occupancy of the GTX480 is 35% while the one of Quadro FX5800 is only 26%. The occupancy of a resource is therefore a good indicator to predict the AVF variation trend for a single application executed on different chips. However, it does not provide information on the actual AVF value. In fact, benchmarks with similar occupancy have significantly different AVFs (e.g., backprop and scan in Figure 4.11). This observation confirms that occupancy is not the only contribution to AVF that has to be investigated.

When considering the local memory (Figure 4.12), while the correlation between AVF and the occupancy of the resource still holds, a clear AVF variation trend between the four chips, valid for all benchmarks, cannot be identified. This suggests

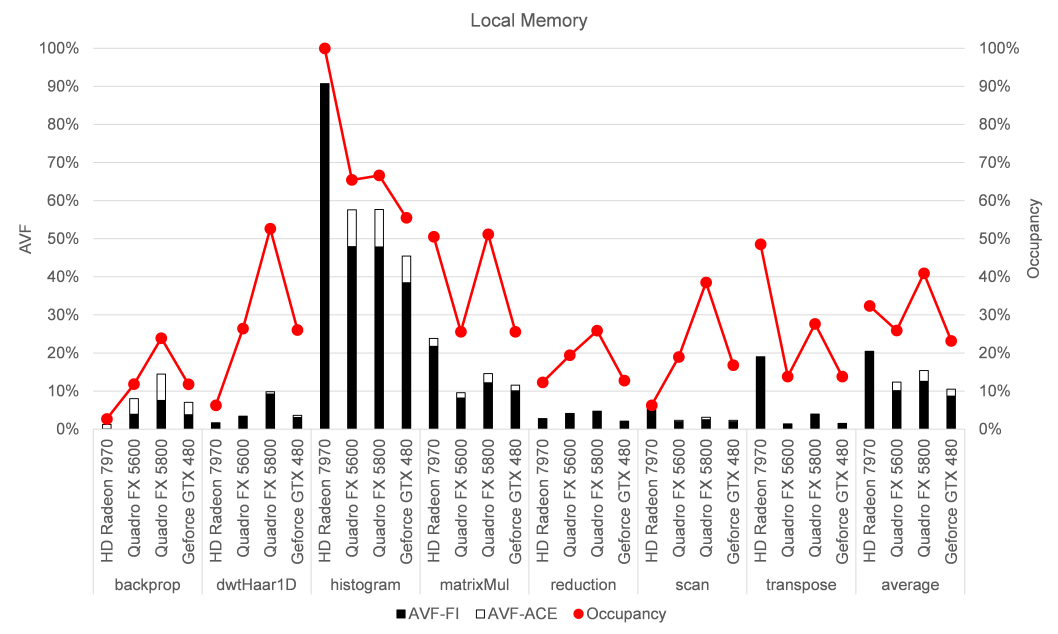


Fig. 4.12 AVF for Local Memory measured by FI and ACE analysis (Source: [2]).

that the way this resource is used strongly depends on the executed application code. In particular, the AVF of HD Radeon 7970 is significantly higher than the other architectures for histogram, where occupancy is 100%. In scan and reduction benchmarks the vulnerability is almost equal for all architectures while Quadro FX 5800 has a slightly higher AVF for backprop and dwt where its occupancy is 22% and 53%, respectively.

To discuss vulnerability decoupled from the resources occupancy we can use the AVF Util. Figure 4.13 reports the AVF Util of the register file while Figure 4.14 the one of local memory. Again, both figures present AVF Util based on both FI and ACE analysis for the considered GPU models. The AVF Util is mainly influenced by the software logic masking and by the different ISAs of the GPU chips. Therefore, it is hard to observe a clear trend or correlation in its value between NVIDIA and AMD architectures. However, it is interesting to note that the three NVIDIA chips, which implement very similar native ISAs and use the same programming model, have very similar AVF Util for each benchmark. Apart from the different local memory and register file size, which in turn influence the number of vulnerable resources leading to different vulnerabilities (we provide more details in Figure 4.16), the NVIDIA GPUs present different wavefront scheduling mechanisms (warps in NVIDIA/CUDA terminology). Each SIMD unit can accommodate a different number of wavefronts.

Changing the number of resident wavefronts changes the scheduling process. This leads to a variation of the vulnerability timing windows for both the registers and the memory words. An increment of the time a wavefront has to wait before being scheduled leads to a longer exposure of a critical resource to a fault. Moreover, the number of wavefronts that a single CU can concurrently execute is another factor that influences the wavefront scheduling. The CUs of NVIDIA Quadro FX 5600 and NVIDIA Quadro FX 5800 can schedule a single wavefront at a time while the CUs of NVIDIA GeForce GTX 480 process two wavefronts in parallel. This difference is highlighted by ACE analysis results for AVF Util of register file (Figure 4.13), which measure reliability on the basis of the vulnerable timing windows of the used memory elements. This does not apply to local memory since it is shared among all the wavefronts of the same compute unit. In details, for some benchmarks, the two NVIDIA Quadro chips report very close values while the NVIDIA GeForce chip shows a different value in benchmarks gaussian, histogram, kmeans and transpose.

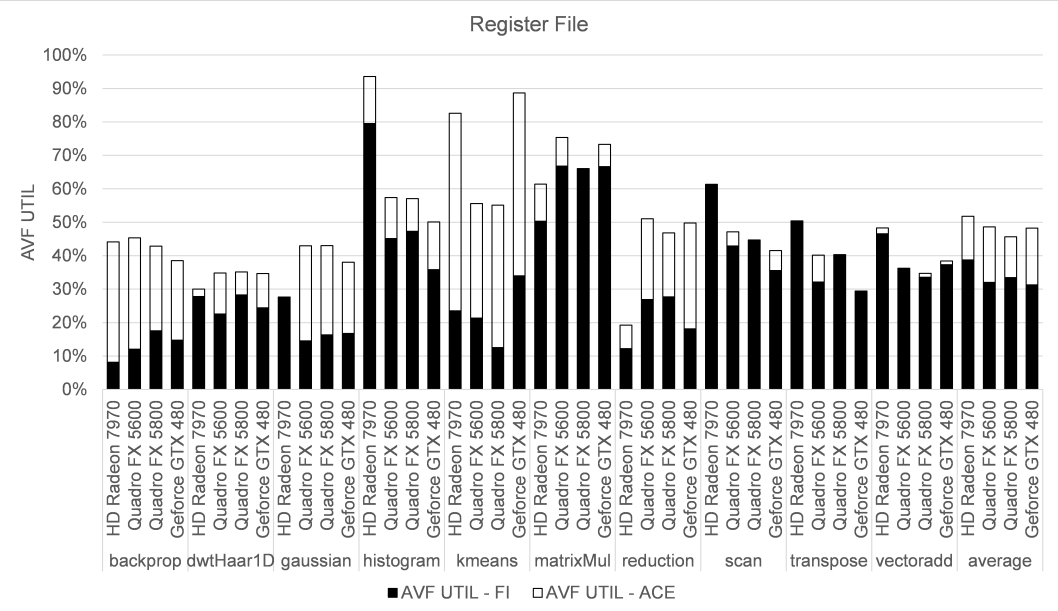


Fig. 4.13 AVF Util for Register File measured by FI and ACE analysis.

Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14 also allow us to compare AVF estimations obtained with different techniques, i.e., FI and ACE analysis. Such a comparison, must consider two main aspects: the measurement accuracy and the time required to perform the analysis. Regarding the accuracy, it is well-known from the literature that the error margin and the confidence interval of statistical fault

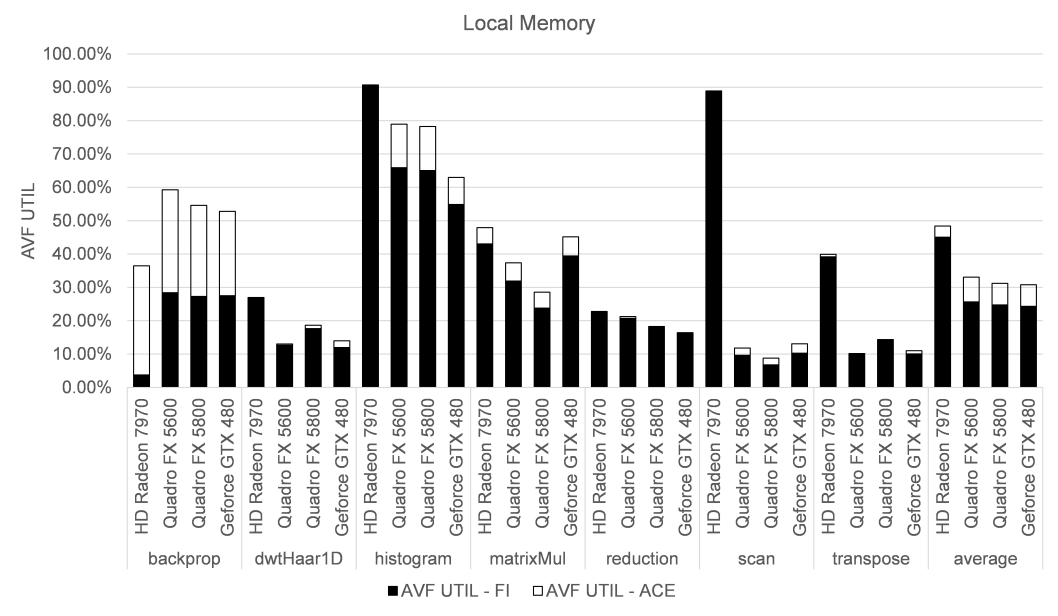


Fig. 4.14 AVF Util for Local Memory measured by FI and ACE analysis.

injection are determined according to the number of injected fault. Differently form FI, the accuracy of the ACE analysis cannot be quantified even if it is well known that it delivers pessimistic evaluations. This trend is confirmed for the register file, where FI estimates lower AVF compared to ACE analysis (Figure 4.11 and Figure 4.13). The error is strongly benchmark dependent. In particular, in our ACE analysis, we do not consider the program logical masking of faults. Figure 4.11 shows that ACE analysis AVF overestimation is lower for the local memory than for the register file. This can be explained since the register file and local memory are used in a different way by the work-items of an application. The only exceptions to this trend are histogram and backprop. For these two benchmarks, the ACE analysis introduces a higher error. This can be explained looking at the AVF Util (Figure 4.14). Their AVF Util significantly changes depending on the evaluation method (FI or ACE analysis). Although, the error between AVF Util based on FI and AVF Util based on ACE analysis is higher for backprop than histogram, histogram occupies more intensely the local memory compared to backprop. Thus, even if histogram features the second highest difference in local memory AVF Util (for different methods of evaluation), its high local memory occupancy leads to the highest difference in AVF depending on the evaluation method. Among the different benchmarks, backprop, is the one presenting the highest AVF overestimation between FI and ACE analysis for both register file and local memory, respectively 3.3 and 1.1 times higher (Figure

4.11, Figure 4.12). On average, the overestimation of the AVF and AVF Util made by ACE analysis with respect to FI is respectively 95% and 80% for the register file, while 15.9% and 17.4% for the local memory. It is also interesting to remark that, for some combinations of benchmarks and architectures, we observe that ACE analysis slightly underestimates AVF. This applies to HD Radeon for dwtHaar1D (0.35 p.u), histogram (0.38 p.u) and reduction (0.93 p.u) and to Quadro FX 5800 for reduction (0.31 p.u). Finally, in case of scan we observe a singularity: ACE analysis underestimates vulnerability barely. However, this difference is very close to the 2.88% error margin of fault injection, which estimates AVF Util equal to 88.9% against 85.2% of ACE analysis.

In terms of simulation time, the single-run ACE analysis offers significantly better performance compared to FI. Table 4.2 quantifies this benefit comparing the simulation time of ACE analysis with the number of fault Injections Per Hour (IPH) that we were able to simulate for each benchmark employing both GUFI and SIFI. However, it is important to remember that this benefit must be traded-off with the reduced accuracy delivered by ACE analysis and with the capability of FI to precisely quantify the error margin of the computed metrics. Nevertheless, looking at the results provided Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14, we can conclude that, despite its lower accuracy, ACE analysis is still a useful method for having an idea about the vulnerability of a hardware component or the differences between benchmarks in a shorter time than fault injection. Given this consideration, in the remaining of this section discussions will focus on results obtained resorting to fault injection experiments.

Figure 4.15 combines the AVF of the different hardware structures with the raw bit soft-error rate of the technology considering an intrinsic λ of 1mFIT per bit. The result is the global FIT of the GPU (λ_S) defined in Equation 4.3. The figure breaks down the contribution of the register file and local memory to the global λ_S. Interestingly, the contribution of the local memory is significantly lower than the one of the register file for most benchmarks and in some cases, it is null: gaussian, kmeans and vectoradd. This is expected since the kernels of these three workloads do not use local memory to exchange data among work-items of the same work-group. Although, both size and vulnerability of a structure affect the FIT rate, it is the size that seems to be the predominant factor. In our experimental setup, the size of the register file is bigger than the one of the local memory and we observe a higher number of failures caused by faults in the register file for all benchmarks

Table 4.2 Simulation time required to perform the reliability analysis.

Benchmark	**Multi2Sim**		**GPGPU-Sim**	
	ACE time (s)	**IPH**	**ACE time (s)**	**IPH**
backprop	3	1200	13	277
dwt	9	400	1	3600
gaussian	29	124	37	97
histogram	173	21	44	82
kmeans	24	150	90	40
matrixMul	21	171	20	180
reduction	4	900	4	900
scan	5	720	2	1800
transpose	2	1800	6	600
vectoradd	39	92	5	720

and architectures except for histogram. In this case, the local memory AVF is much higher than the one of the register file and represents the main contribution to the FIT. The GPU chips can be ordered based on their local memory size as follows: Quadro FX 5600 and Quadro FX5800 (both 16 KB), GTX480 (48 KB) and HD Radeon 7970 (64KB). This order reflects on the trend we observe in the fraction λ_S due to faults in the local memory (Figure 4.15). Only backprop does not follow this trend because the local memory AVF for HD Radeon 7970 is negligible (0.10%).

The GPU architectures whose hardware component sizes are larger show higher values of FIT, as reported in the average case. In fact, HD Radeon 7970, which has twice the memory elements of GeForce GTX 480, has the highest value of FIT, while the smaller Quadro GPUs feature low FIT. This trend can be explained by the fact that larger GPUs usually require a larger amount of memory elements in order to exploit better their intrinsic parallelism. Executing more work-items concurrently increases the number of potentially vulnerable resources[10]. Figure 4.16 visualizes this trend: it shows for each benchmark and architecture the number of memory elements (bits) used in the context of the computation (vulnerable resources).

The bigger size of a GPU hardware component naturally makes it more vulnerable to soft errors. However, it increases the execution parallelism and thus improve performance. Therefore, as discussed in Section 4.4, to combine the reliability evaluation with the performance profile of each benchmark and GPU chip we analyze the EPF (Figure 4.17) and the IPF (Figure 4.18) metrics because FIT alone (Figure

[10]vulnerable resources: total number of memory elements (bits) used during the entire execution

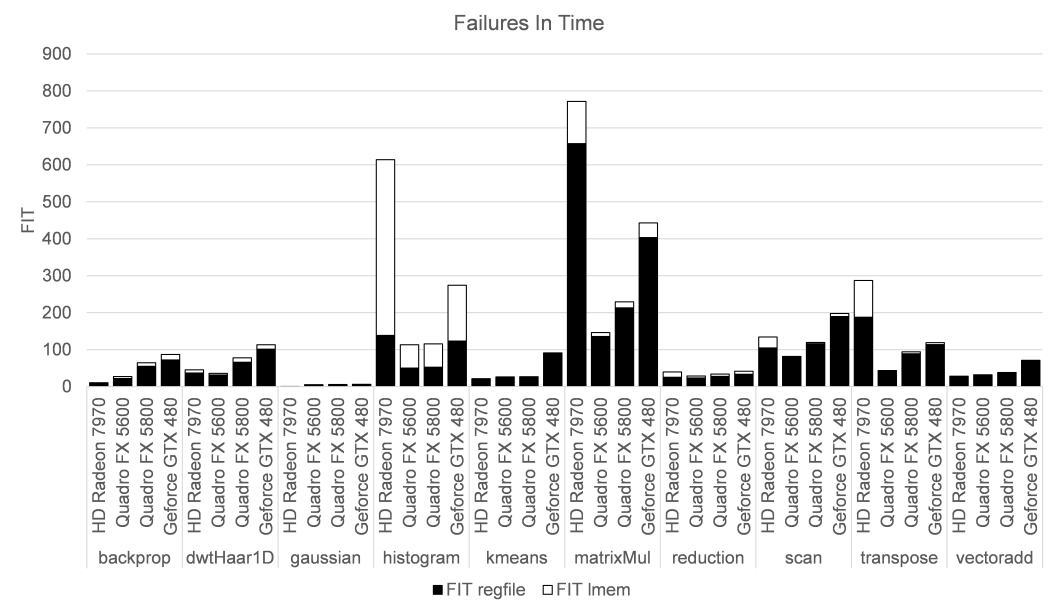


Fig. 4.15 Breakdown of Failures in Time rate using the AVF measurements from Fault Injection.

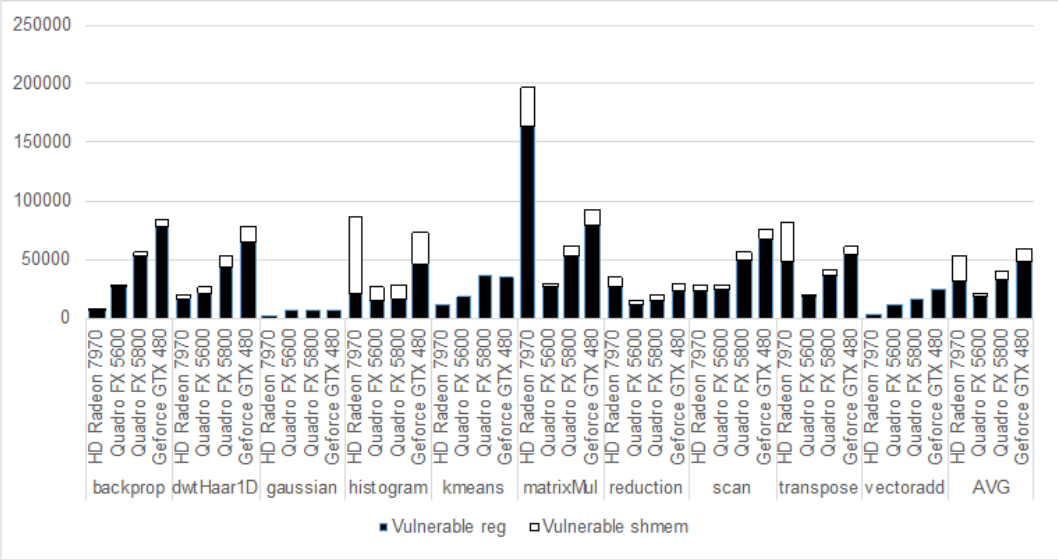


Fig. 4.16 Vulnerable resources in bits.

4.15) does not take into account the amount of work carried out by the GPUs before a failure arises. EPF incorporates the execution time and FIT for a program, while IPF also includes information about the instruction throughput of GPUs when executing

an application. Table 4.3[11] summarizes for each benchmark (rows) and architecture (cols) the execution time (cycles) as well as the number of executed instruction required to compute EPF and IPF.

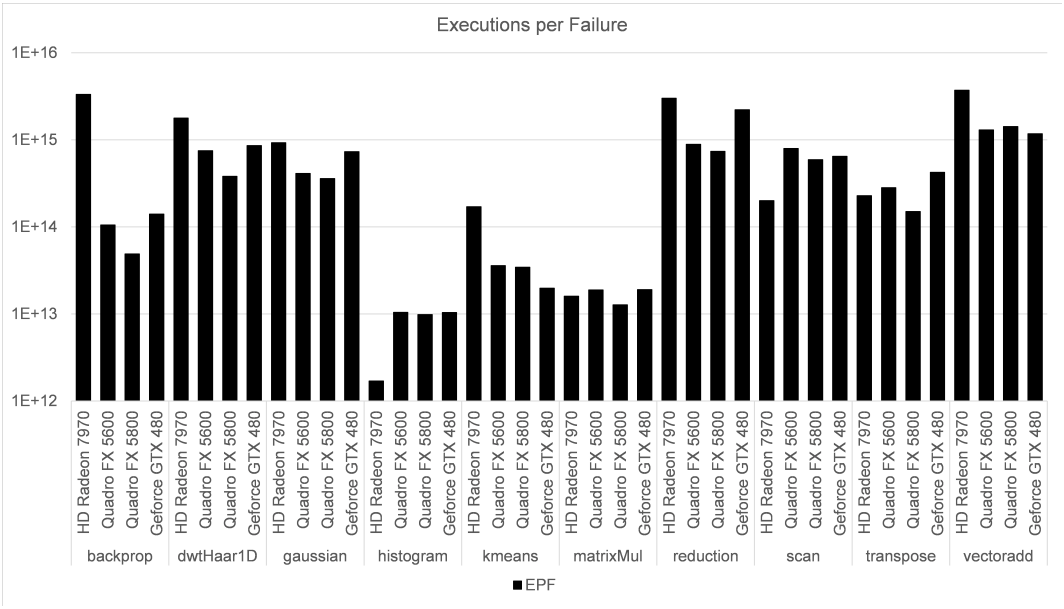


Fig. 4.17 Executions per Failure (EPF) (Source: [2]).

As reported in Equation 4.6, the IPF for a particular benchmark is proportional to the EPF and to the instruction throughput. Since this throughput strongly depends on the target execution device, to fairly compare different GPU architectures we must look at the EPF instead of IPF while the IPF is useful for evaluating the reliability of different programs on the same GPU chip. In one hand, the EPF metric is useful to the architects who can quantify the effectiveness of a hardware based error protection technique which can be applied to their designs (if needed) along with a performance cost at the early design stages. Larger EPF numbers show a larger number of executions before a failure and different protection mechanisms can deliver different improvements in the FIT rates and can also have different impact on performance. Combining performance and reliability measurements in the EPF metric delivers a broader view for decision-making. This could be for instance

[11] About NVIDIA GPU frequency, GPGPU-Sim the width of the pipeline is equal to the warp (wavefront) size. To compensate for this, the SIMT Core Cluster clock domains are adjusted. For example the super pipelined stages in NVIDIA's Quadro FX 5800 (GT200) SM running at the fast clock rate (1GHz+) are modeled with a single slower pipeline stage running at 1/4 the frequency. Thus, a 1.3GHz shader clock rate of FX 5800 corresponds to a 325MHz SIMT core clock in GPGPU-Sim [84]

Table 4.3 Execution time and instructions of each benchmark.

Benchmark		**HD Radeon 7970**	**Quadro FX5600**	**Quadro FX5800**	**GeForce GTX480**
	freq	925 MHz	337.5 Hz	325 Hz	700 MHz
backprop	cycles	94376	423594	369855	206834
	inst.	2108160	10312032		
dwt	cycles	41072	44998	39412	25859
	inst.	1839075	1180042		
gaussian	cycles	5862543	561060	555732	541687
	inst.	7308189	5488224		
histogram	cycles	3198537	1031394	1029746	885491
	inst.	20029440	21784328		
kmeans	cycles	913526	1278216	1267144	1397604
	inst.	31930960	35984844		
matrixMul	cycles	269591	439591	400594	299346
	inst.	10924032	15007744		
reduction	cycles	27836	47377	47086	27231
	inst.	312736	854719		
scan	cycles	123763	18707	16572	19721
	inst.	3025801	468720		
transpose	cycles	50862	98911	82821	49942
	inst.	733184	2818048		
vectoradd	cycles	31687	29219	21603	30225
	inst.	1523712	638976		

important when evaluating real-time applications that are not continuously executed, but they are scheduled once every time period. On the other hand, IPF is useful to the programmers who want to quantify the effectiveness of software redundancy based protection techniques which can be applied to their programs running on the same architecture, thereby enchaining the error resilience of their applications at a performance cost. IPF summarizes both the performance cost and the resilience improvement. In Figure 4.17, the HD Radeon has the highest EPF for benchmarks backprop, dwtHaar1d, gaussian, kmeans, reduction, scan, transpose and vectoradd while histogram has almost the same EPF for Quadro FX 5600 and GTX480. This applies also to matrixMul. In Figure 4.18, HD Radeon, backprop and gaussian have higher IPF than the other benchmarks while Quadro FX5600, Quadro FX 5800 and GTX 480 have the highest IPF in the case of gaussian.

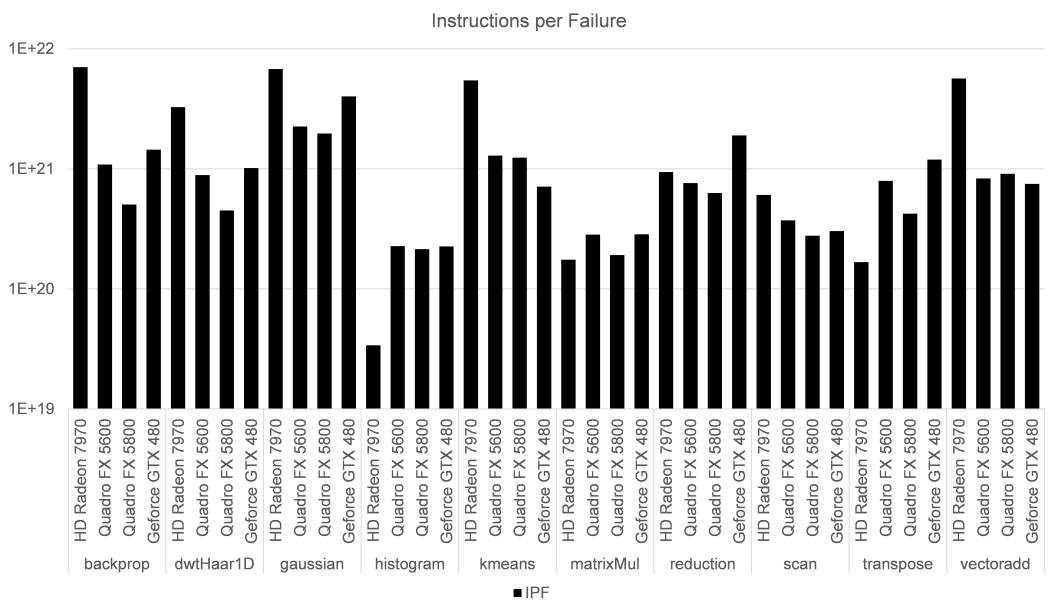


Fig. 4.18 Instructions per Failure (IPF).

In summary, our reliability and performance analysis has identified the value of many different measurements for reliability evaluation of GPUs. The AVF Util metric, for example, nicely describes the capability of a workload to mask soft-errors, while IPC and the number of vulnerable resources impact reliability in space and temporal dimensions. When software-based fault tolerance techniques are considered, the ratio between IPC and the vulnerable resources is a relevant aspect. This ratio is particularly important since it can be controlled to some extend by compiler optimizations as well as by the software designers.

4.6 Conclusions

This chapter addresses reliability evaluation in the context of GPGPU. Two of the most popular techniques for reliability estimation of CPUs were adapted to GPUs: fault injection and ACE analysis. A reliability framework for the AMD Southern Islands GPU architecture is introduced: SIFI. Experimental results demonstrated the capability of SIFI to evaluate reliability and to help systems engineers in the exploration of the design space, thus enabling system optimization. More specifically SIFI allows to explore the design space by analyzing different microarchitectural

configurations. The evaluation can be computed according to several reliability metrics such as AVF, AVF Util, FIT, EPF and IPF.

SIFI was also employed to compare AMD and NVIDIA GPU architectures. This analysis highlighted the most relevant relationships concerning the GPU microarchitectures, reliability and performance. In this context, EPF and IPF, two new metrics, are proposed enabling to jointly evaluate reliability and performance.

SIFI can analyze the vector register file, the scalar register file and the local memory of the AMD Southern Islands GPU architecture. Further investigations can focus on the extension of SIFI benefits to other hardware structures and GPU architectures as well.

Finally automated design space exploration could be addressed in order to deliver a tool capable of finding autonomously the best trade-off among the system requirements.

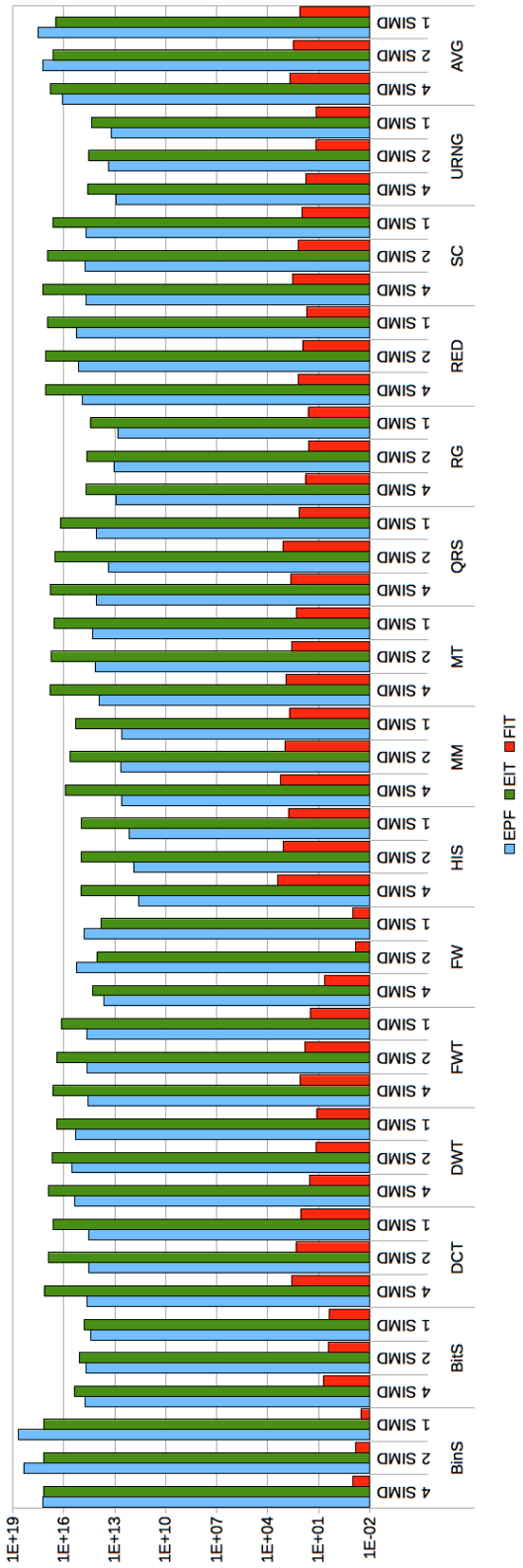


Fig. 4.19 Joint comparison of reliability and performance: the Execution Per Failure changing the number of SIMD Units per CU.

Chapter 5

A Bayesian model for reliability evaluation of CPU-based systems

The previous chapter introduced reliability evaluation. Two of the most employed methodologies were presented, fault injection and ACE analysis. They aim at estimating reliability by means of simulations. This approach can be very slow, in the case of fault injection, or slightly inaccurate, in the case of ACE analysis. This chapter introduces a novel methodology based on a statistical approach, trying to deliver a fast and accurate analysis. In contrast to Chapter 4, the reliability analysis proposed here targets CPUs, however this approach can be easily extended to GPUs by introducing some changes. This statistical reliability analysis is able to estimate system reliability considering both the hardware characteristics of the CPU and the executed software, in presence of hardware transient and permanent faults. The hardware resources of the processor and the instructions of program traces are employed to build a Bayesian Network. Finally, the probability of input errors to alter both the correct behavior of the system and the output of the program is computed. To prove the effectiveness of the proposed methodologies reliability was analyzed for a set of benchmarks chosen from the MiBench suite[92] executed on an x86-64 CPU model. Experimental results obtained by fault injection were compared with the ones obtained by the proposed Bayesian. University of Athens carried out the fault injection experiments and the extraction of the software traces. The proposed Bayesian Network model is able to provide accurate reliability estimations in a very short period of time. As a consequence it can be a valid alternative to fault injection. Part of this chapter was previously published in [3].

5.1 Introduction

Reliability is an important design aspect for computer systems due to the aggressive technology miniaturization [93, 94, 15], as previously discussed in this thesis. Unreliable hardware components affect computing systems at several levels. Raw errors can manifest due to several causes such as physical fabrication defects, aging or degradation (e.g., NBTI), process variations, environmental stress (e.g., radiations). Raw hardware errors can propagate through other layers of the system (e.g., architecture, software) up to the output. During the propagation process, raw errors can be masked by each of the affected layers.

A significant effort in the research community has been spent to analyze masking properties at technological and architectural level [95], [96]. Moreover, understanding the effect of software on the reliability of a complex system in which unreliable hardware is present is also gaining increasing importance. The software has intrinsic masking capabilities that can be enhanced by the implementation of software level fault tolerance mechanisms [97], [98] and [99]. However, these mechanisms often incur in a significant performance overhead. Therefore, the role of the software stack coupled with the target hardware architecture must be carefully considered when system reliability is analyzed.

Several studies focus on understanding and modeling how hardware faults can propagate and manifest through a software application, without considering how these faults can actually propagate and be masked within the software [100–103]. A very important contribution that examines the impact of software on the architectural vulnerability factor (AVF) of a system is provided in [104]. The paper defines a so called Program Vulnerability Factor (PVF), isolating the software-dependent (architecture-level masking) portion of the AVF from the hardware-dependent (microarchitecture-level masking) portion. This metric captures the architecture-level fault masking inherent in a program, allowing software designers to make quantitative statements about programs resilience to soft errors. PVF can be measured using an architectural simulator, a dynamic binary translator such as Pin [105] or resorting to ACE-like analysis [106]. Moreover, a comprehensive PVF calculation is affected by the software workload that may increase the simulation effort. Another interesting contribution has been proposed in [107] where a statistical model is proposed to estimate the capability of a software application to mask hardware errors. The main contribution of this chapter is to introduce a statistical

model that simply requires a preliminary characterization of the hardware masking probability and then it is able to analyze software applications executed on this hardware. However, this work does not take into account execution time so that it may lead to inaccurate estimations especially for big programs. Overall, one of the main limitations of the publications presented so far is that they are limited to the analysis of the effect of soft errors: permanent and intermittent faults are not considered at all.

The proposed methodology introduces a new statistical approach for the estimation of the reliability of a microprocessor-based system considering both the hardware and the software layer. The software execution on the microprocessor is modeled in the form of a Bayesian Network that describes relations among resources (e.g., registers, memory elements, functional units) involved in the execution of the instructions composing a program. The Bayesian model is then exploited to compute a probability of correct (error-free) execution of the software in the presence of a hardware fault, used to estimate the overall reliability of the system. To construct the model, a preliminary characterization of the Instruction Set Architecture (ISA) of the microprocessor is required. This characterization aims at evaluating the probability of successful execution of each instruction of the target ISA in presence of faults in the microprocessor hardware blocks. Transient, intermittent and permanent faults can be considered in this phase without affecting the way the high-level model is constructed.

Bayesian Networks represent a successful model employed for reliability estimates in different fields [108]. In the software engineering domain, they were employed to model software reliability in the distributed domain (Kishore at al. [109], etc.). A few publications consider the application of Bayesian Networks to model system reliability in hardware devices as well [110], [111]. However, they do not consider the interaction of hardware and software in an instruction-based environment. In addition, full system reliability, considering technology, hardware and software layers, is modeled by Bayesian Networks in [4] and it is discussed in the next chapter of this thesis.

To validate the proposed statistical model we analyzed 4 different MiBench benchmarks [92] executed targeting permanent and transient faults on top of a x86-64 microprocessor. Experimental results highlight that reliability estimations are accurate when compared to those obtained by time-consuming fault-injection

experiments performed using a microarchitectural fault injector [112]. At the same time, the proposed approach enables a significant reduction of the time required to perform the reliability analysis, enabling fast and accurate reliability evaluations.

5.2 Reliability analysis

Bayesian Networks (BNs) are an efficient statistical model to represent multivariate statistical distribution functions. They can model relationships among random variables and their respective probability density functions by means of conditional probability functions. In particular, conditional dependencies are expressed by a *Direct Acyclic Graph* (DAG). Nodes of the DAG represent conditional probability functions of the random variables, while edges represent conditional dependencies among random variables. If two nodes are connected, it means that the random variables they represent are conditionally dependent.

The proposed statistical reliability analysis methodology focuses on the estimation of the probability of failure of a program running on a specific microprocessor. Bayesian networks are employed to model program traces, i.e., sequences of instructions issued by the microprocessor when the program is executed with a specific workload. In particular, the information required to build the BN is the execution time and the involved hardware resources of instructions. Several traces can be obtained from a single program by profiling its execution with different workloads using a dedicated profiler. This work employs [107], but other profilers can be used as well without affecting the estimation model. The collection of the analyzed traces represents different program behaviors that may generate different error masking effects during the program execution. Program traces are sequential lists of instructions that can therefore be efficiently modeled in the form of a Bayesian Network whose main constraint is the absence of loops in the network. As traces are evaluated, the system failure probability can be estimated by averaging the trace analysis results.

To assess the system reliability, both the hardware and the software layers must be taken into account. In the proposed model the microprocessor ISA represents the direct link between the two layers. In particular, each instruction is considered with regard to the hardware resources required for its execution. To build a BN model of a program trace running in a system, it is essential to take into account the hardware resources, error sources and the instructions of the analyzed trace.

Hardware resources, *Res*, are divided into two subsets: the storage resources subset (S_{res}), which includes registers and memories, and the computational resources subset (C_{res}), which includes functional units required for computation. Resources can be affected by errors during program execution. Consequently, when dealing with reliability analysis, it is fundamental to define an error model. For soft errors the exponential distribution is assumed (See Section 2.4). The reliability function of a resource ($R_{res}(t)$), i.e., the probability of an error-free resource in a period of time t, is given by the following equations:

$$\lambda_{res} = \lambda_{comp} \frac{A_{res}}{A_{comp}} \tag{5.1}$$

$$R_{res}(t) = e^{-\lambda_{res} t} \tag{5.2}$$

where λ_{comp} is the SER of the electronic component dependent on the target technology, while λ_{res} is the resource SER assuming equal spatial distribution of error occurrence in the component. Equation 5.1 can be useful when the failure rate is defined for the entire hardware resource and the failure rate of just a portion is needed (i.e., an entire ALU and a single multiplier or the entire resister file and a register). Since a single resource is part of a bigger electronic component, it can be assumed that its reliability is related to the one already defined for the component. More specifically, λ_{res} is a portion of λ_{comp} and it is proportional to the fraction of its silicon area A_{res} over the total area of the component A_{comp}. It is worth mentioning that Equation 5.1 is consistent with Equation 2.4 presented previously. For permanent errors, the stuck-at-fault model is addressed. If a permanent fault occurs in a resource, it means that an internal signal or the output signal of the affected resource has a fixed value.

The occurrence of both transient and permanent errors in hardware resources can be masked at the hardware level due to several masking effects. We therefore consider a masking probability for each instruction of the ISA, representing the probability that an instruction prevents an error occurrence to propagate. In this work, each instruction I is characterized by a set of input storage and computational resources, $R_{I_{in}} \in S_{res} \cup C_{res}$, required for the computation, and a set of output storage resources, $R_{I_{out}} \in S_{res}$, that are updated by I. All resources in $R_{I_{in}}$ can mask errors during the execution of an instruction and are therefore characterized by a given masking probability. Masking of hardware resources can be obtained by error

protection mechanism implemented in the design (e.g., ECC for memories, Triple Module Redundancy for registers, etc) or by data transformations they perform (e.g., an AND functional block can mask a bit flip of an operand if the not-faulted operand is a 0).

Computing the masking probability can be performed according to two strategies: *operand* analysis or fault injection. The former consists of analyzing the mathematical operations in which resources are involved. In most of the cases this operation is simple and it is not time consuming. On the contrary, the latter method requires a bigger effort. In fact, a fault injection campaign is required to analyze the error resiliency of the system resources. Errors can be injected at RTL level as well as at gate level. However this operation must be done just once, since when masking probability of a resource is known, it can be employed in all the system the resources it is part of.

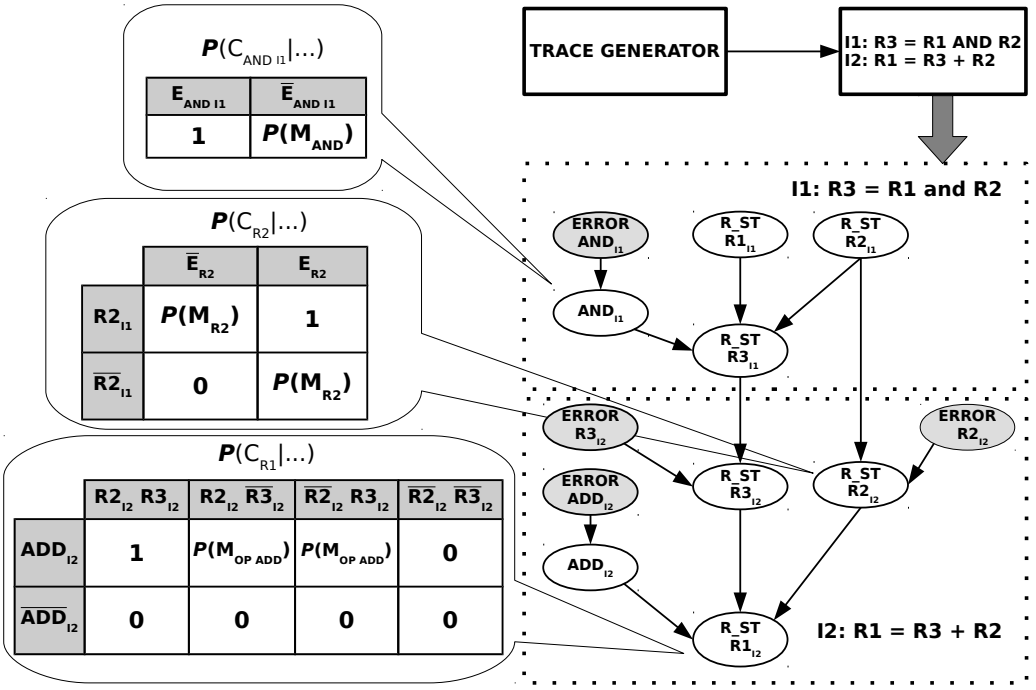


Fig. 5.1 Example of a bayesian network model for a simple sequence of two instructions (Source: [3]).

To understand how to build a BN model, let us consider the simple program trace composed of two instructions that is reported in Fig.5.1. The first step is to identify

those resources belonging to $R_{I_{in}}$ and $R_{I_{out}}$ of each instruction of the trace. In fact, they represent the nodes of the BN. Each node N can assume two possible states: error-free (denoted as N) or faulty (denoted as $\overline{\text{N}}$).

In the example of Fig.5.1, $I1$ has $R_{I_{in}} = \{R1, R2, AND\}$ and $R_{I_{out}} = \{R3\}$, while $I2$ has $R_{I_{in}} = \{R3, R2, ADD\}$ and $R_{I_{out}} = \{R1\}$. We therefore introduce 8 nodes in the network: AND_{I1}, $R1_{I1}$, $R2_{I1}$, $R3_{I1}$, ADD_{I2}, $R3_{I2}$, $R2_{I2}$, $R1_{I2}$. Once resource nodes are defined, resource dependencies are modeled by means of network edges. Resources belonging to $R_{I_{in}}$ are connected to resources belonging to $R_{I_{out}}$ of the instruction. Moreover, output resources of an instruction may be connected to input resources of a following instruction to model instruction dependency. At this point the BN topology is ready and error nodes, representing the input of the network can be therefore included. Error nodes, when faulty, express the raw probability that an error occurs in a resource. According to our model, input errors can only affect resources belonging to $R_{I_{in}}$.

Each node is then associated to a set of conditional probabilities that quantify the probability of correctness of the node depending on the correctness of the input nodes. Four cases must be considered.

Input nodes Input nodes of the network can be either error nodes or nodes associated to storage resources for which the initial error probability is known. A single probability of correctness is associated to these nodes. In case of soft errors, computational resources can be affected by errors while they are employed. Consequently, the probability of a error-free error node connected to a computational resource is computed according to (5.2) with a proper λ_{res} and t equal to the execution time of the instruction. On the other hand, storage resources can be affected by external errors during the period of time elapsed between a write and a read operation. Therefore, for error nodes connected to storage resources, the probability of an error-free node is calculated by means of (5.2) considering a proper λ_{res} and t equal to the period of time between the read and the write of the resource. On the opposite, in case of hard errors, the probability of an error-free error node can be one or zero, depending on the presence of the permanent error regardless the kind of resource the node is connected to.

Nodes identifying computational resources in $R_{I_{in}}$ of an instruction (e.g., AND_{I1} in Fig.5.1) These nodes have a single incoming edge connected to an error node. Two conditional probabilities must be defined as reported in Fig.5.1 for these nodes where $P(M_{AND})$ identifies the masking probability of the hardware resource by means of protection mechanisms implemented in the design (i.e., ECC for memories, TMR, etc.). Since the masking probability of protection mechanisms is decided during the design phases and it is decided according to the system requirements, RTL implementation of the circuit and fault injection are not necessary.

Nodes identifying storage resources in $R_{I_{in}}$ of an instruction (e.g., $R2_{I2}$ in Fig.5.1) For these nodes there are two possible causes of faults: (1) the resource was already corrupted in a previous instruction or (2) an error corrupts the resource in the time interval between the execution of two instructions. In this case, four conditional probabilities must be defined as reported in Fig.5.1 where $P(M_{R2})$ identifies the masking probability of the hardware resource. It is worth to highlight that, in our example, we always consider that, when more than one input node is faulty, the probability of correctness of the current node is zero. This is a worst case assumption that however reduces the complexity of the characterization of the masking probability of each resource.

Nodes identifying storage resources in $R_{I_{out}}$ of an instruction (e.g., $R1_{I2}$ in Fig.5.1) These nodes may have several inputs. Therefore, the number of conditional probabilities to set is equal to the number of possible combinations of states the input nodes may assume. Similar to the previous case, we consider that errors can be masked only when computational resources in $R_{I_{in}}$ are error-free and at maximum a single storage resource in $R_{I_{in}}$ is faulty. In the example, we denote with $P(M_{OPADD})$ the masking probability that the ADD operation performed by the computational resource will mask errors in a input storage resource. This probability is commonly known as logic masking and it can be computed by simulating the behavior of each instruction with different combinations of operands in input [107]. The masking probability is obtained by injecting faults into input operands and comparing the obtained output with the one without faults. This operation only targets operands and it does not involve the gate and the RTL implementation of the circuit.

Once conditional probabilities are set up properly for every node, the BN can be solved and the probability of correct execution of a trace can be estimated. However, only a subset of the instructions of a trace directly affect resources that identify the final outcome of the computation. We denote this subset of instructions as active state instructions (A_I). To compute the probability that a program trace is correct, error probability is taken into account only for output resources of the active state instructions. We define such resources as active resources (A^{res}) and we denote with A_i^{res} the subset of active resources modified by instruction I_i . Given these definitions, the probability for the active instruction I_i to be correct ($P(I_i)$) can be computed as the probability that all its active resources are correct:

$$P(I_i) = \prod_{R \in A_i^{res}} P(R) \tag{5.3}$$

since the probabilities of correctness of output resources of an instruction are statistically independent.

A program trace T is correct (error-free) if all active state instructions are correct. The probability of correctness of a trace ($P(T)$) can therefore be computed as:

$$P(T) = P(\bigcap_{I \in A_I} I) = \prod_{I \in A_I} P(I|J, \forall J < I) \tag{5.4}$$

When computing $P(T)$, we need to consider that the event that all active instructions are correct is not statistical independent. We therefore need to multiply the probability of correctness of every active instruction given that all its previous active instructions are correct, $P(I|J, \forall J < I)$. This is possible by setting the evidence in the Bayesian Network that all its previous active instructions are correct. The network is then solved and $P(T)$ can be computed.

Once the probability of correctness of a trace is computed, for permanent faults, this value is equal to the final masking probability of the system, $P(M_{System})$. Instead, for transient faults some additional computation is required. In fact, the error rate of the system while executing the trace can be computed by inverting Equation 5.2, thus obtaining:

$$\lambda_{estimated}^{T} = -\frac{ln(P(T))}{t_T} \tag{5.5}$$

where t_T is the execution time of the analyzed program trace. To obtain more accurate estimates of the system failure rate, λ_{BN}, a simple or a weighted average

can be applied to all $\lambda_{estimated}$ of the analyzed traces. Weights of the traces can be set according to the probability that a trace is executed by the analyzed system:

$$\lambda_{BN} = \sum_{T \in analyzed\,traces} \lambda^{T}_{estimated} \times w_i \tag{5.6}$$

where $\sum w_i = 1$. Finally, to compute the masking probability of the system:

$$P(M_{System}) = 1 - \frac{\lambda_{BN}}{\lambda_{comp}} \tag{5.7}$$

5.3 Experimental results

This section presents the results obtained by implementing the presented Bayesian network model in a reliability analysis tool, and by applying it to a test program executed on a given microprocessor architecture.

5.3.1 Framework implementation

We implemented a complete automatic framework able to perform the reliability analysis described in Section 5.2. The framework is composed of two main modules: (i) the *trace generator*, and (ii) the *Bayesian network analyzer*.

The trace generator (developed by University of Athens) is built on top of the MARSSx86 [113] full system, cycle-accurate architectural simulator. It simulates the execution of the target program on the x86-64 architecture. During the execution, a detailed trace of the list of executed instructions, and for each instruction the list of resources (e.g., physical register or memory virtual addresses) that are involved in the execution is generated.

Generated traces are analyzed by the Bayesian Network analyzer in order to build the Bayesian model of the trace and to estimate the related failure rate. The Bayesian Network analyzer is built on top of SMILE [114], a free C++ framework for the analysis of Bayesian models. It is important to highlight here that, when analyzing the network of a real application composed of hundreds of thousands of instructions, the size of the related network may increase up to a level that saturates the available computational resources. To overcome this limitation, thanks to the conditional

probability offered by BNs, the network of the trace is split into several sequential subnetworks each depending on the probabilities computed by the previous network. By applying this iterative approach, scalability of the reliability analysis on complex applications can be achieved with very limited computation time.

5.3.2 Experiment setup

To validate the proposed reliability estimation methodology we set up four case studies. They consist of software applications, chosen among the MiBench benchmarks [92], running on the x86-64 architecture. They are *qsort* (32 traces), *aes* (32 traces) and *sha* (16 traces) for transient errors, and *sha* (16 traces) for permanent errors. For each experiment, several traces are analyzed. For a preliminary analysis, hardware faults are just injected into microprocessor physical registers belonging to the Integer Register File.

Traces are extracted resorting to the MARSSx86 simulator [113] employed in the Fault Injector tool described in Subsection 5.3.2. The tool has the ability to trace various microarchitectural events (such as committed instruction sequence, memory access pattern) which can be reassembled to build the actual execution trace of a program.

To validate the proposed Bayesian Network analyzer, the same benchmarks with the related workloads are analyzed resorting to the proposed Bayesian model and resorting to an extensive architectural fault injection campaign. Computed results are then compared to evaluate the accuracy and the performance of the proposed model. In detail, the reliability is expressed in terms of masking probability of the system (Equation 5.7).

Bayesian model

In order to set conditional probabilities for BN nodes some preliminary operations must be performed. First of all, instructions of the x86-64 architecture must be analyzed. Masking probabilities are evaluated according to the *operands* analysis explained in Section 5.2. Secondly, since the proposed Bayesian model addresses the ISA and faults are injected at microarchitectural level, some precaution must be adopted. This operation requires an analysis of the physical system to be evaluated.

In our experiments faults are only injected into the physical register file instead of ISA registers. To overcome this issue we decided to adopt a strategy to tune input error probabilities for ISA registers. In the x86-64 architecture a register rename table keeps the data regarding the renaming process of each physical register into an ISA register. When permanent faults are addressed, the input error probability of the faulted resource is set to the probability that the register is mapped to the faulted physical register instead of being set to one.

$$P(Error)\frac{1}{\#\_of\_PHYS\_REGs} \tag{5.8}$$

For transient errors, as register renaming is dynamic, we assume that the number of physical registers that are involved in the computation at the same time is equal to the number of architectural registers. In other words, we assume that the area of the architectural register file, A_{ARCH_RF}, can be evaluated as:

$$A_{ISA\_RF} = A_{PHYS\_RF} \times \frac{\#\_of\_ISA\_REGs}{\#\_of\_PHYS\_REGs} \tag{5.9}$$

where A_{PHYS_RF} is multiplied by the number of ISA registers over the number of physical registers. Moreover, we assume that all registers have the same size. As a consequence the area of an architectural register, A_{isa_reg}, is the ratio between the number of ISA registers and A_{ISA_RF}.

Finally, for each program trace, we considered as active state instructions those instructions storing in a memory element the result of the computation for the last time, based on the results of the trace generator.

Fault Injection

Fault injection experiments have been performed using the MaFIN [112] fault injector built on top of MARSSx86 [113] full system, cycle-accurate architectural simulator. MaFIN is capable of injecting single and multiple transient (bit-flip), intermittent (stuck-at-0, stuck-at-1), permanent (stuck-at-0, stuck-at-1) faults, or a mixture of them to the microarchitectural structures of the x86-64 microprocessor. Furthermore, MARSSx86 allows to monitor the propagation of a hardware fault to the upper level of system stack: the application output. The extracted output files can be analyzed to classify the injected faults: when no mismatch at the application output is detected,

the application execution is labeled as correct. As already mentioned, University of Athens took care of performing fault injection experiments and of generating program traces.

We performed a statistical fault injection campaign, on MaFIN (see Table 5.1). Using [89] (see Equation 4.12), we compute a fault population for 99% confidence level and 3% error margin. The calculation leads to a total of 1843 different injections on the integer physical register file. For transient faults the bit-flip is placed in a randomly selected position (i.e., a bit in a physical register) at randomly selected clock cycles. Instead, for permanent faults the choice of the faulty bit and the logic value of the stuck-at is random.

Finally, results of fault injection are usually expressed in terms of AVF, however for this comparison they are converted to masking probability: $P(M_{System}) = 1 - AVF$.

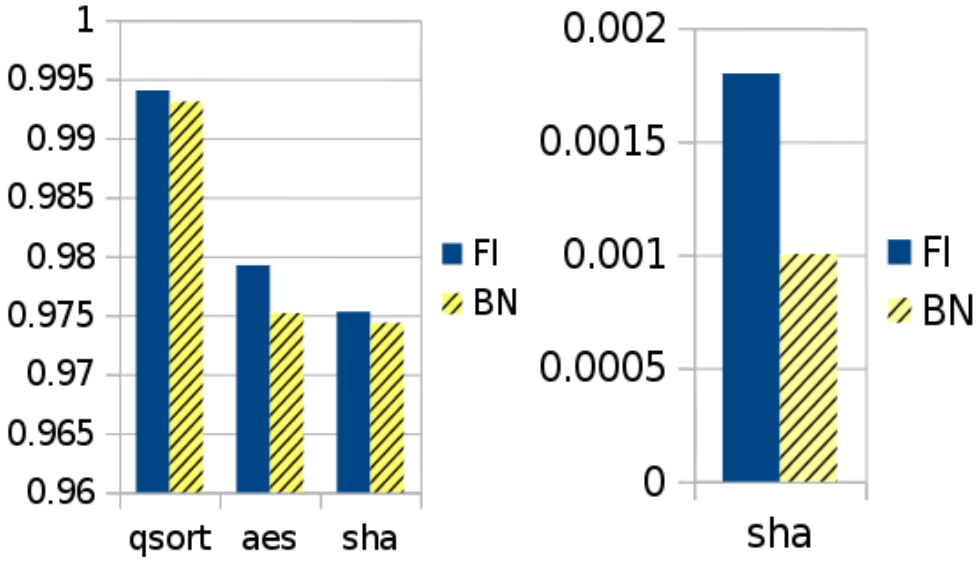
Table 5.1 Marssx86-64 microprocessor model configuration.

Parameter	**Setting**
Fetch/Issue/Commit	4/4/4 instructions per cycle
Combined Predictor	16KB (64K entries, 2 bits/entry, 16 bits BHR)meta pred.: 64K entries
Physical Register File	256 INT; 256 FP; 16 Store; 24 Branch
Reorder Buffer	128 entries
Functional Units	4 clusters (ALUs: 2 INT, 2 FPU; 4 AGUs)
Cache Memories	L1-D (32KB, 4-way, WB) L1-I (32KB, 4-way, WB) L2 (1MB, 16-way, WB)

5.3.3 Results

Reliability estimate of the analyzed program traces are compared for the fault injection and the Bayesian model. Figure 5.2 compares accuracy of the two methods.

We can state results show that Bayesian model estimations are very close to the FI ones. In particular, they belong to the uncertainty range of 3% according to [89].



(a) Transient faults

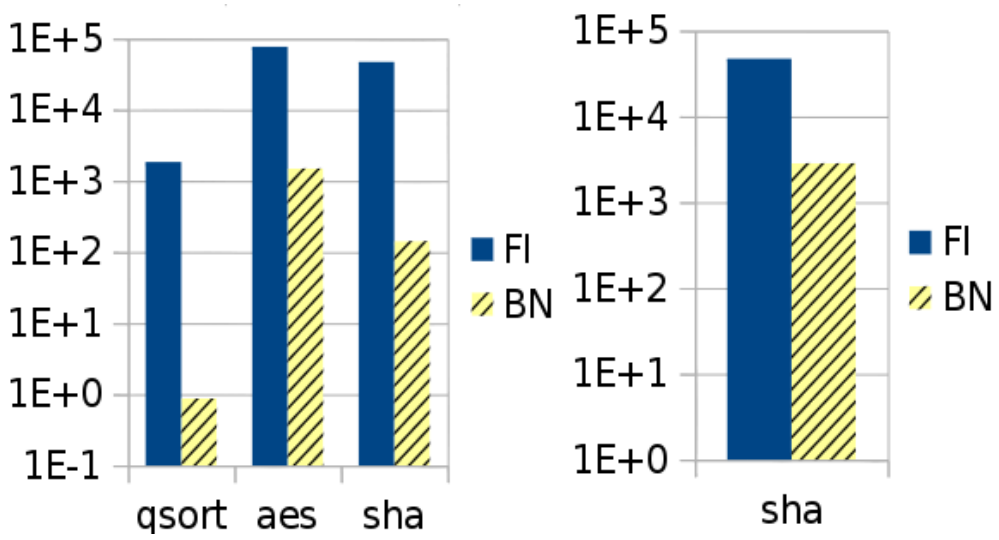
(b) Permanent faults

Fig. 5.2 Estimations of masking probability for both the Bayesian Network and the Fault Injection approaches (Source: [3]).

Finally, Figure 5.3 compares simulation time for each experiment. The figure clearly shows that, resorting to the statistical model, estimation time is reduced by several orders of magnitude thus enabling very fast estimations.

5.4 Conclusion

The methodology presented in this chapter proposes a new statistical approach for the estimation of the reliability of a microprocessor-based system, taking into account the interaction between the hardware and the software layer. Preliminary experimental results performed on the MiBench benchmarks clearly show that the proposed approach is able to provide accurate and fast estimations when compared to a similar analysis performed using micro-architectural level fault injection. The ability of providing fast reliability evaluations, considering both the hardware and the software layer, is a key feature to enable optimized designs, leading to a progressive avoidance of common practices employed to reach high reliability levels, such as worst case design.



(a) Transient errors (b) Permanent faults

Fig. 5.3 Timing performance comparison of simulation time of a single trace for both the Bayesian Network and the Fault Injection approaches. Time is expressed in seconds (Source: [3]).

The proposed Bayesian model can be easily adapted to other instruction set architectures as the only requirement is a profiler able to track the sequence and the execution time of every instruction performed by the target system. In order to reach a comprehensive reliability statistical analysis, further investigations can address the modeling of microarchitectural resources other than the register file, as the ROB, the LSQ and the BTB.

Chapter 6

Reliability estimation of complex digital systems

System reliability estimation during early design phases facilitates informed decisions for the integration of effective protection mechanisms against different classes of hardware faults. When not all system abstraction layers (technology, circuit, microarchitecture, software) are factored in such an estimation model, the delivered reliability reports must be excessively pessimistic thus leading to unacceptably expensive, over-designed systems. To overcome this lack, we present a system reliability framework adopting a cross-layer approach built on top of a Bayesian model. The methodology proposed in Chapter 5 introduced Bayesian Networks, however it was a preliminary study to show the potential of Bayesian models in the context of reliability assessment for digital systems. The previous chapter targeted just the CPU and considered only faults appearing in the register file, while this chapter extends the reliability analysis to the system as a whole. Moreover, the model presented by this chapter introduces the possibility of identifying the weakest components of the system in terms of reliability, thus enabling system optimization by means of Bayesian reasoning (more details about the optimization process are presented in the next chapter). We propose a scalable, cross-layer methodology and supporting suite of tools for accurate but fast estimations of computing systems reliability. The backbone of the methodology is a component-based Bayesian model, which effectively calculates system reliability driven by the masking probabilities of individual hardware and software components considering their complex interactions.

To evaluate the usability, applicability and scalability of the developed model, methodology and tools, both a set of benchmarks and real life applications were analyzed.

Part of the work presented in this Chapter was previously published in [4, 115]. This methodology was developed in the context of the 7th Framework Program of the European Union through the CLERECO Project, under Grant Agreement 15 611404. The suit of tools addressing the technology, hardware and software layers were developed respectively by "Universitat Politecnica de Catalunya" (UPC), "University of Athens" (UoA) and "Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier" (LIRMM) and integrated in the system level framework described in this thesis.

6.1 Introduction

When looking at system-level reliability, failing to meet a reliability requirement may add excessive re-design costs to recover and may have severe consequences on the success of a product [116]. Worst-case design with large margins to guarantee reliable operation has been employed for long time. However, it is reaching a limit that makes it economically unsustainable due to its performance, area, and power costs [117].

Hardware faults may propagate through the hardware (HW) and software (SW) layers of the system stack, reaching the system output, or be masked during this propagation process. Different protection mechanisms can be employed at different layers implementing what is nowadays called cross-layer reliability enhancement [37, 118, 115]. Accurately measuring the impact on system reliability of any change in the technology, circuit, microarchitecture and software is a complex design task, involving design teams from all abstraction layers. The task has multiple objectives because reliability must be traded-off against other crucial design attributes such as performance, power, and cost [119]. Unfortunately, tools and models for cross-layer reliability analysis are still at their early stages compared to other very mature design tools (e.g., performance and power optimization tools). The current practice of industrial standards is to rely either on time-consuming gate-level fault injection campaigns or on simplified models that guarantee smaller computation time but

deliver very coarse-grain and conservative (i.e., pessimistic) reports of the system reliability [85, 86].

This Chapter proposes a novel system-level cross-layer reliability assessment framework built on top of a component-based Bayesian model of the target system. In component-based system reliability modeling, the system is more than the sum of its parts. Each component affects globally perceivable properties of the entire system. By carefully integrating parameters from all layers of the system stack we are capable of accurately evaluating the failure rate of the full system. The proposed system-level reliability model and supporting tools deliver several key contributions with respect to current approaches. The target system can be described in terms of components and the model can efficiently describe how faults and errors propagate through components, accounting for complex interactions among them that are not modeled with simpler combinatorial models (e.g., reliability block diagrams or fault trees [120]). Components can be unplugged from the system during their characterization and the effect of their interaction can be recombined later, thus enabling reuse of information. Moreover, the model is highly parameterized and scalable. It enables including any factor that can potentially affect the reliability of the system (e.g., environmental factors such as location and temperature) by simply adding new variables to the model. The model scales efficiently to complex systems with an analysis time that is significantly shorter than traditional fault injection while maintaining adequate levels of accuracy. Other system-level approaches provide similar execution time [85, 86] but their accuracy is significantly lower and can thus lead to more costly design decisions for reliability. A statistical model itself would be useless for reliability assessment in real applications without supporting instruments to populate the model for a specific system and workload. Along with the model, we therefore present a complete framework comprising a tool-chain able to compute the FIT of the final system, based on the proposed system-level reliability model. In this chapter, we address transient faults (soft errors), but if tools and models to estimate conditional failure probabilities for different classes of faults (i.e., intermittent and permanent faults) are developed, the proposed model can be used to study their effect as well.

The remainder of this Chapter is organized as follows: Section 6.2 discusses the state-of-the-art solutions adopted for architectural and software reliability evaluation as well as the most popular Bayesian models adopted to assess reliability. The proposed reliability framework is introduced by Section 6.3 and the developed tools

by Section 6.4. Then, results are presented and analyzed in Section 6.5. Finally Section 6.6 concludes this chapter.

6.2 Related works

6.2.1 Architectural level reliability analysis

The Architectural Vulnerability Factor (AVF) of a microprocessor and its estimation has attracted significant attention by the research community. As discussed previously, the AVF of a hardware structure is the fraction of faults in it that affect the correct program operation [15]. Most AVF estimation methods are based on offline analysis with architecture or RTL level simulators [15, 121, 86]. Offline AVF estimation is a complex process, requiring major modifications to the simulators and many resources to track values and instructions as they travel through microprocessor components. Only a limited number of instructions (short programs) can be analyzed in a reasonable amount of time because of the excessive memory requirements of ACE (Architectural Correct Execution) analysis. Online or real-time AVF estimation has been also presented [122–126], but it still requires heavy offline simulation and calibration for different workloads. It is not clear to what extent the parameters calibrated for one set of workloads will give accurate estimation for another set. A general drawback shared by all these methods is their AVF over-estimation due to worst-case assumptions. A 7x AVF over-estimation is reported in [85], whereas [86] reports that even a refined ACE-based analysis (which requires even more elaborate modifications of the microprocessor simulator model) leads to up to 3x over-estimation. This leads to over-designed systems. The model and method we propose in this work aim to contribute to the design of computing systems without excessive costs for reliability.

6.2.2 Accounting for software effects in reliability analysis

In [127] the authors discussed a first attempt to perform static analysis of a computer system including its software. However, the approach is limited to errors affecting the instruction opcodes before they enter the microprocessor pipeline. Three seminal papers by Sridharan and Kaeli first considered the software layer in the reliability

assessment of a system [104, 128, 16]. They propose to compute a Program Vulnerability Factor (PVF) to quantify the portion of AVF that is attributed to a user program. This concept has been further extended in [22] with the introduction of the concept of the System Vulnerability Stack. The System Vulnerability Stack is a significant advance towards the definition of a system reliability model accounting for all layers of the system stack. However, it is over-simplified and it considers that layers are statistically independent from each other, to allow computing the global AVF as a simple product of the vulnerability factors of each layer. This is obviously not the case in a real system in which there is a very intricate interaction among the different layers and among components of each layer, and this approximation leads to pessimistic predictions.

Another interesting solution that considers the impact of the application software running on embedded microprocessors was discussed in [107]. Despite the fact that it provides promising results, the method is still limited to transient faults in embedded microprocessors. Moreover, being based on static analysis of code traces, it does not capture important masking effects introduced during dynamic execution.

6.2.3 Bayesian models for reliability estimation

Bayesian Networks models are very useful statistical models employed in many disciplines. Weber et al. in a comprehensive review report more than 200 papers published between 1998 and 2008 in international journals on applications of Bayesian Networks in different fields, i.e., dependability, risk analysis and maintenance [129]. Differently from state-space based models such as Markov models [130] Bayesian models are better suited when component-based modeling is required.

Among the different application fields, Bayesian models have been largely used to create software reliability models, i.e., to predict the probability of failure-free software operation for a specified period of time in a specified environment [131–134]. Software reliability differs from system reliability considered in this thesis since it reflects the software design perfection, rather than hardware manufacturing perfection and tolerance of the system to design variability and environmental stresses [135].

Finally, a large set of publications present high-level theoretical Bayesian models to predict system reliability in different fields ([136–138] and their references). The

main drawback of these approaches is that they focus on models optimizations to improve the capability of the analysis but do not provide solutions to collect all parameters required to build the model in a specific application domain.

Our model and reliability assessment method is a major step forward. We explicitly consider a Bayesian model in the field of cross-layer reliability focusing on system level effects of hardware faults. Our methodology comprises both the theoretical framework required to properly describe the target system using a Bayesian Network and a complete integrated framework of tools able to compute all parameters required to feed the model for virtually all realistic cases of hardware and software components. Moreover, it introduces the possibility to identify the weakest components in terms of reliability, thus enabling reliability optimization of the system under design.

6.3 The proposed system level reliability framework

In this work, we focus on system reliability assessment against soft errors. Errors resulting from low-level faults may manifest, be masked or be propagated through the HW and SW layers of the system stack, possibly resulting in partial or total failure of the system activities (Figure 6.1). Other reliability issues such as HW/SW design bugs are out of the scope of this work.

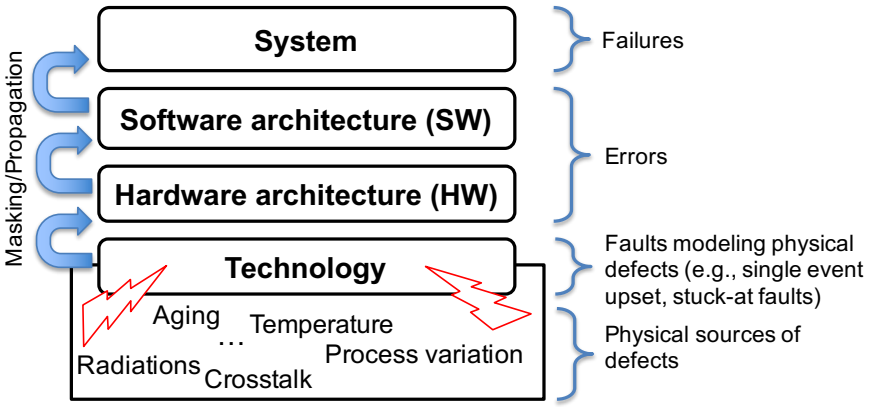


Fig. 6.1 The system stack: faults originate at the lower layer of the system stack and are either masked or propagated to the upper layer possibly resulting in a failure at system level (Source: [4]).

We propose a reliability assessment model allowing designers to obtain reliability estimations early in the system design cycle. This supports architectural decisions and gives indications about those portions of the system that are critical and deserve customized development effort to improve reliability. Among different modeling styles [130], following the component-based system design approach we propose a component-based reliability model [139]. In component-based reliability modeling, system reliability is estimated using reliability information and other properties (e.g., size, complexity, etc.) of individual system components and their interconnections (the system architecture). Our model exploits Bayesian Networks as a statistical foundation for full-system reliability analysis. BNs offer several interesting features for system reliability modeling: (i) efficient calculation scheme, (ii) capability of fitting on field and simulation data, (iii) intuitive and compact representation, (iv) decision support. A BN is a compact representation of a multivariate statistical distribution function encoding the Probability Density Function (PDF) governing a set of random variables by specifying a set of conditional independent statements together with a set of conditional probability functions.

The proposed system reliability model is composed of a qualitative model representing the architecture of the system and a quantitative model, representing the reliability of each component and their relations.

6.3.1 Qualitative model of the system

The system architecture is defined through a *directed acyclic graph G* (Figure 6.2):

$$G = (V = (C \cup P), E = (RR \cup PR)) \tag{6.1}$$

The set of vertices V is split into two subsets: components (C) and parameters (P). Components are blocks composing the system. Depending on the architectural layer (technology, HW, SW) the component definition changes as discussed later in this section. Components, $c_i \in C$, are associated to Bayesian nodes, i.e., their reliability is associated to a set of random variables. Parameters, $p_i \in P$, are special vertices that are not direct part of the system Bayesian model. They represent implementation details of a component (e.g., operating temperature, workload, etc.) exploited by our framework to build the quantitative model of the system described later in this section. The $RR = (c_i, c_j) \in C \times C$ (reliability relations) are the set Bayesian arcs

that define temporal or physical reliability relations among components, e.g., a failure state of a component may influence the state of another component. Finally, the $PR = (p_i, c_j) \in P \times C$ (parameters relations) are the set of relations between a component and its implementation parameters. Based on the system stack shown in Figure 6.1, components of a system are split into four subsets or domains (Figure 6.2) each requiring different techniques to be characterized for reliability.

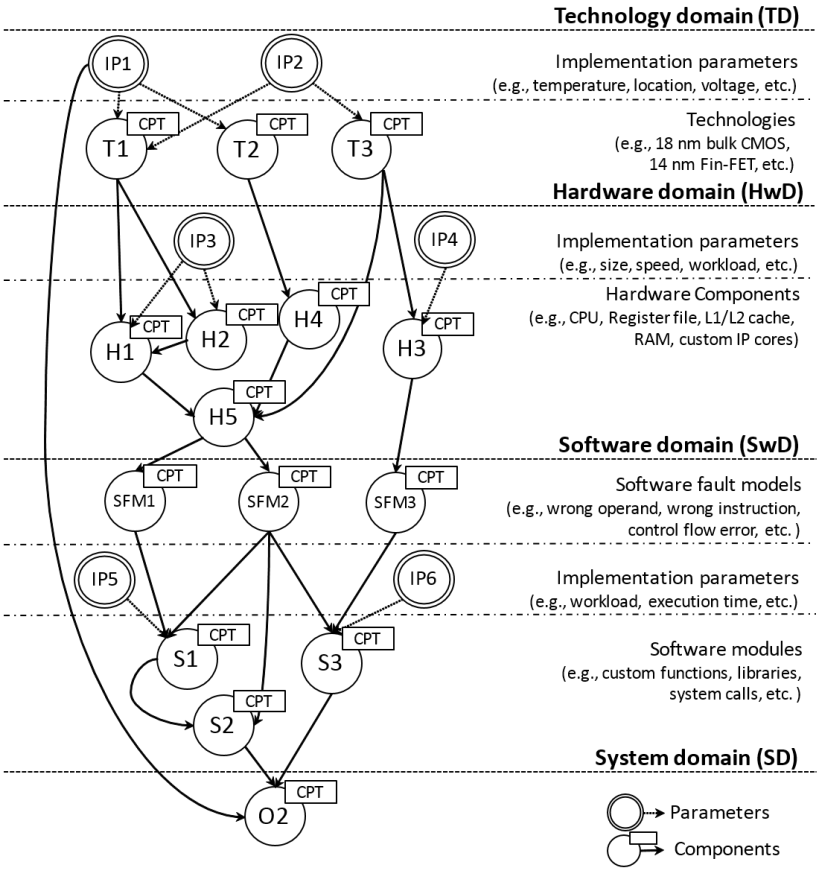


Fig. 6.2 System reliability estimation model. System components are modeled by component nodes. The topology of the network provides the qualitative description of the system. Conditional probability tables (CPT) associated to each component node of the network provide the quantitative description of the reliability of the system. Parameter nodes model information required to compute the CPTs of the component nodes (Source: [4]).

The *technology domain* (TD) models the physical layer of the system. Components in this domain list all fabrication processes used to build the hardware structures of the system (e.g., 16nm Bulk CMOS for a microprocessor component, 14nm FinFET CMOS for external DRAM, 14nm NAND Flash for external storage, etc.). These components set the raw fault probabilities of the system. Implementation parameters in this domain model physical quantities influencing the raw fault probability of a technology (e.g., temperature, voltage, location, radiation effects, etc.).

The *hardware domain* (HwD) models the hardware architecture. Components in this domain list the hardware blocks such as CPUs, GPUs, memories, accelerators, custom IP cores, used to build the system. Granularity at which hardware blocks are modeled in this domain depends on the level of detail the designer needs for the reliability analysis, and the degree of freedom the designer has with the design of the selected components. A complex component such as a microprocessor can either be considered as a whole or split into its subcomponents (e.g., register files, ALUs, buffers, queues, speculation units, etc.) to allow a fine-grained analysis and optimization. Each hardware component is associated to a set of implementation parameters such as size (e.g., number of bits of a memory), speed, workload, etc.

The *software domain* (SwD) models the software architecture. To decouple the analysis of the SwD from the one of the HwD, special attention is required to define the interface between the two layers. Components of this domain are further split in two sub-domains: (1) software fault models (SFMs), and (2) software modules (SMs).

SFMs are the approach introduced by our model to translate hardware failures into the software domain and therefore decouple the two domains so that the corresponding supporting tools for the hardware and the software domains can operate independently. SFMs model program alterations that can be linked to alterations of the Instruction Set Architecture (ISA) of the hardware block executing the software. Table 6.1 lists the set of SFMs currently supported by our framework for selected microprocessor architectures. The table is not intended to be exhaustive. Additional SFMs can be plugged in the model given that proper tools for the evaluation of their occurrence and effect are designed.

On the other hand, SMs model the software architecture. The granularity of the description in this domain depends on the specific application. A node may represent

Table 6.1 Example of SFMs taxonomy.

Software Fault Model	**Description**
Wrong Data in a Operand (WDO)	An operand of the instruction changes its value
Source Operand Forced Switch (SOFS)	An operand is used in place of another
Instruction Replacement (IR)	An instruction is used in place of another
Faulty Instruction (FI)	The instruction is executed incorrectly
Control Flow Error (CFE)	The control flow is not respected

a high-level library or framework, a single function, a portion of a function or even a specific data structure (e.g., an array), thus allowing for a fine-grained customization of the model. Implementation parameters in this domain mainly include the workload of each module.

Performing system level reliability analysis requires the definition of a set of observation points where the behavior of the system is evaluated and properly classified. In most applications, observation points are a set of software components whose outputs define the outcome of the system. Nevertheless, associating the concept of observation points to the software domain is limiting. To model this concept, the *system domain* (SD) has been split from the other domains and placed at the higher-level of our model to separately identify those components where the entire system's reliability is observed.

6.3.2 Quantitative model of the system

The quantitative model of the system defines the probability of occurrence of an error/fault in a component depending on the condition of its direct interacting components and on its implementation parameters. In a Bayesian model such as the one proposed in this chapter, the quantitative model is a set of Conditional Probability Tables (CPTs): there is one CPT, $\Theta_{c|\mathbf{U}}$, for node $c \in C$ whose size depends on its parent nodes $\mathbf{U}$ (Figure 6.3).

Each node c is associated to a set of states, which identify possible error or error-free conditions of the node (e.g., a memory can be error free, or it can be

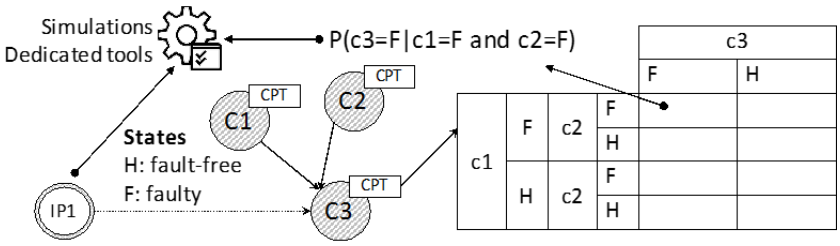


Fig. 6.3 Example of CPT for node c3 (Source: [4]).

affected by a single bit-flip, or by a double bit-flip). The set of states of the nodes depends on the node domain and the specific characteristics of the node. For each state of a node, we need to look at all combinations of states of its parent nodes. Each such combination is called an *instantiation* $\mathbf{u}$ of the parent set $\mathbf{U}$. The CPT $\Theta_{c|\mathbf{U}}$ maps each instantiation $c \mid \mathbf{u}$ to a probability $\theta_{c|\mathbf{u}}$ such that $\sum_i \theta_{c|\mathbf{u_i}}$. Nodes without parents, called root nodes, are described according to their marginal probability distributions.

Computing CPTs can be both difficult and time consuming. It is typically an assignment given to a group of specialists that need to collect information and organize them according to the model. In this chapter, we perform a significant step forward by proposing a framework of integrated tools, each designed to operate in one of the four node domains and able to compute conditional probabilities for major classes of software and hardware modules of modern electronic systems. The tools enable to consider each instantiation of a node and to setup a set of simulations to compute the corresponding conditional probabilities in a fast and optimized way.

In details, each node is characterized by the following output states according to its domain:

- Technology domain: faulty or not-faulty
- Hardware domain: one output state for each error classification of the hardware component
- SFMs: refer to Table 6.1
- Software domain: one output state for each Software Faulty Behavior (SFB). In the proposed model Error-free, SDC and Unresponsive are the considered SFBs (Table 6.2), however the model can integrate other SFBs.
- System domain: one output state for each system failure mode.

Table 6.2 Software Faulty Behavior Classifications.

Fault Classification	Description
Masked	The application execution terminates normally. All the application outputs are correct.
Silent Data Corruption (SDC)	The application execution terminates normally. However, the application outputs are different from the fault free outputs.
Unresponsive	The application execution does not terminate normally: it stops working or it never stops.

It is important to note that, if a node has many parents or if the parents can take a large number of states, the CPT becomes very large. This issue is particularly severe at the SwD, where the size of the CPTs of software nodes (expressed in terms of number of columns) is equal to:

$$\#CPT\ colums_{direct} = \#sfb^{\#parent\ nodes} \tag{6.2}$$

This equation reports that the CPT size grows exponentially with respect to the fan-in of the nodes. As a consequence the size of the CPTs explodes even if a node has a relatively small number of parents. In our model, we consider 3 SFBs. Considering the contribution of the parent nodes, the size of the CPT is in the order of billions of columns in case of 20 parents (3^{20} conditional probabilities). Consequently, this approach can be just applied to analyze simple programs.

To cope with the exponential number of probabilities in the CPTs two solutions can be adopted. The first consists of resorting to the Noisy-MAX approach [140]. The Noisy-Max is a generalization of the interaction of a child node and its direct ancestors that allows reducing the size of the computed CPT thus reducing the number of required simulations.

```
/* l_p  |          BENIGN          |             SDC          |         UNRESPONSIVE      |
 * ------------------------------------------------------------------------------------------
 * r_p  | BENIGN | SDC | UNRESPONSIVE | BENIGN | SDC | UNRESPONSIVE | BENIGN | SDC | UNRESPONSIVE |
 * ------------------------------------------------------------------------------------------
 * BEN  |    1   |  0  |      0       |    0   |  0  |      0       |    0   |  0  |      0       |
 * SDC  |    0   |  1  |      0       |    1   |  0  |      0       |    0   |  0  |      0       |
 * DUE  |    0   |  0  |      1       |    0   |  1  |      1       |    1   |  1  |      1       |
 * ------------------------------------------------------------------------------------------
```

Fig. 6.4 The error propagation policy from parents to child.

The other, instead, consists of merging parent nodes in a k-ary tree. This approach does not lead to any loss of accuracy since it reproduces the policy for the propagation

of errors from parent to child nodes (Figure 6.4), therefore it was the adopted solution. In details the policy is described for the following cases:

- wherever there is no error in the parent nodes, no error is propagated to the child node;
- wherever there is an unresponsive error in at least one of the parent nodes, an unresponsive error is propagated to the child node;
- wherever there is an SDC error in one and only one of the parent nodes, an SDC error is propagated to the child node;
- wherever there is an SDC error in more than one of the parent nodes, an unresponsive error is propagated to the child node;

We define nodes of the k-ary tree as dummy nodes. A dummy node is connected to k nodes. Therefore, the CPT size of dummy nodes is equal to $\#sfb^k$. The maximum depth of the k-ary tree with respect to the number of parents of the node is computed as $\log_k(\#parent\,nodes)$. Assuming that there are at most k^i dummy nodes at depth i the total number of CPT columns of the whole k-ary tree is reduced to:

$$\begin{aligned} \#CPT\,colums_{k-arytree} &= \#sfb^k \sum_{i=0}^{\log_k(\#parent\,nodes)} k^i \\ &= \#sfb^k \frac{1-k^{\log_k(\#parent\,nodes)}}{1-k} \\ &= \#sfb^k \frac{1-\#parent\,nodes}{1-k} \end{aligned} \tag{6.3}$$

To compute the best k to minimize the size of the CPTs we compute where the 1st order derivative of Equation 6.3 is equal to 0, in order to find the minimum:

$$\begin{aligned} &\frac{\partial(\#CPT\,colums_{k-arytree})}{\partial k} = \\ &= \frac{(1-\#parent\,nodes)\#sfb^k(k\ln(\#sfb)-1-\ln(\#sfb))}{1-k^2} = 0 \end{aligned} \tag{6.4}$$

It is interesting to notice that this result does not depend on the $\#parent\,nodes$ since we consider it to be equal or greater than 2. Moreover since $\#sfb^k > 0 \forall k$ its contribution can be not considered. Consequently, to find the best k the equation can

be rearranged as:

$$\begin{aligned} \frac{\partial(\#CPT\ colums_{k-arytree})}{\partial k} &= k\ln(\#sfb) - 1 - \ln(\#sfb) = 0 \\ &\Rightarrow k = 1 + \frac{1}{\ln(\#sfb)} \end{aligned} \tag{6.5}$$

From this equation we derive that the best k is a value between 1 and 2. As a result we decide to adopt $k = 2$, so that we model the tree of dummy nodes as a binary tree.

Next step is to define a rule to establish whether to create or not the dummy binary tree solving the following equation with respect to $\#parent\ nodes$:

$$\begin{aligned} \#CPT\ colums_{k-arytree} &< \#CPT\ colums_{direct} \\ \#sfb^{k}\frac{1-\#parent\ nodes}{1-k} &< \#sfb^{\#parent\ nodes} \\ &where\ k = 2 \end{aligned} \tag{6.6}$$

In case we have $\#sfm = 3$ we obtain than $\#parent\ nodes > 2$. However to keep the number of nodes low we choose $\#parent\ nodes > 4$.

6.3.3 Reasoning on the model of the system

Once the system is described, the proposed reliability model can be used to reason about the its reliability properties.

Bayesian reasoning is a well-known approach and the reader may refer to [141] for further details. Two main types of reasoning are supported. In the predictive reasoning, starting from information about fault causes (i.e., raw technology failure rates) the designer is able to obtain new beliefs about their effect on the system failures, following the forward directions of the network arcs. This enables early reliability analysis of the complete system (FIT evaluation). In the diagnostic reasoning the designer reasons from symptoms to cause (backward direction of the network arcs), i.e., the observation of a system failure updates the belief about the contribution of components to the failure. This allows us to identify weak components that most likely contributed to the failure, in order to drive the reliability design effort toward the most critical components thus optimizing the overall system at the lower cost. Moreover, the model can be used to calculate new beliefs when

new information (evidence) is available. For example, by setting the evidence that a given component is in a given state (e.g., a hardware component is faulty), the model enables to update the belief of the system failure given this new information, as well as to update the belief of the root causes that lead to this component failure.

6.4 Integrated tools framework

This section describes the tools required to populate the quantitative information in the proposed system reliability model. Tools addressing the *Technology Domain*, the *Hardware Domain* and the *Software Domain* were developed respectively by "Universitat Politecnica de Catalunya", "University of Athens" and "Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier" and they are employed for this work. On the contrary, we developed the tool targeting the *System Domain* implementing the proposed methodology. A brief description of the tools employed here is reported in the following subsections to give a complete overview of the proposed framework.

6.4.1 Technology Domain

In the technology domain, for different technologies, a tool chain developed by "Universitat Politecnica de Catalunya" was employed to populate the CPTs of the technology nodes. This tool chain is able to characterize the main building blocks composing a logic circuit in order to compute their marginal fault probability with respect to a given failure model[1]. The current implementation focuses on soft errors caused by particle strikes. Figure 6.5 summarizes the technologies and blocks analyzed so far for soft errors. Further technologies and blocks can be analyzed given the availability of a proper technology and circuit model. Each block and each technology is analyzed for different combinations of run-time parameters. Supported parameters currently include combinations of voltage, temperature and geographical location. The geographical location is considered to accurately predict the error rates caused by particle strikes as detailed in Appendix A.

[1]The marginal distribution of a random variable is the probability distribution of the variable without reference to the values of the other variables (i.e., opposite to conditional probabilities). Since technology nodes are root nodes they are described by marginal probabilities rather than conditional probabilities.

Technology (CMOS)	Technology node
Bulk Planar (ASU PTM Models)	22nm and 16nm
Bulk FinFET (ASU PTM Models)	20nm and 14nm
SOI Planar (UTSOI Model)	22nm

X

Circuits
SRAM Cells 6T/8T/10T
Flip Flop - D
Latch
Logic Gates (AND, OR, NOT)

Fig. 6.5 Building blocks and technologies analyzed (Source: [4]).

Figure 6.6 summarizes the simulation workflow. A Python simulation engine drives an exhaustive design space characterization for a given component implemented in a given technology. The analysis is organized into a set of nested loops to simulate an element subject to different operating temperatures, voltage, locations and technology models. To study the impact of particle strikes, in the inner loop, the current injected in the sensitive nodes of an element is iteratively increased until a flip or glitch is detected measuring the stored value (SRAM) or the output (Logic Gates). Each electrical simulation is performed in HSPICE. The minimum charge generated from a pulse that causes a malfunction is stored and defined as the *Qcrit* of that element (see Subsection 2.4). Finally, for each *Qcrit*, a raw soft error rate (SER) is computed using the model in [18].

Through the use of this framework, a technology library of SERs for different blocks under different technologies, geographical locations, and voltage/temperature was built. Data from this library can be fed to our system model any time a new system must be analyzed, without repeating the underlying simulations. The reader may refer to [142] for additional implementation details.

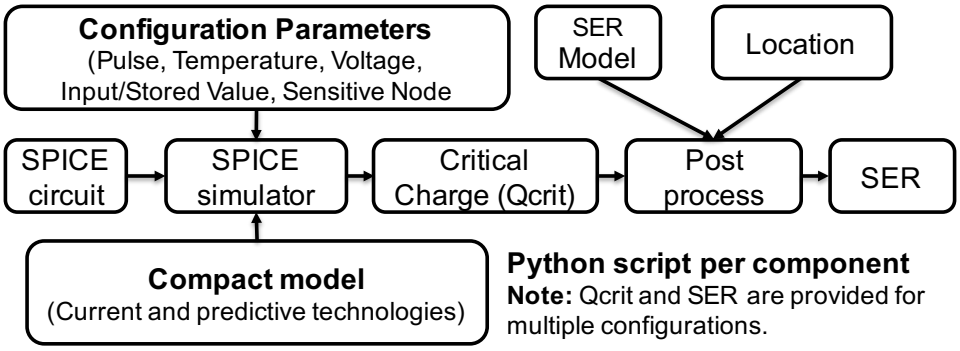


Fig. 6.6 Technology Domain characterization workflow (Source: [4]).

6.4.2 Hardware Domain

At the hardware domain, GeFIN [143], a tool developed by "Univesity of Athens", was employed to characterize the CPTs of the hardware nodes. This tool chain characterizes the different micro-architectural blocks of a microprocessor. Microprocessors are the main target since they are one of the most complex and important blocks of a system. The analysis performed by GeFIN starts from the assumption that a fault (e.g., a SEU) affects one of the blocks of the microprocessor. Whether this fault transforms into an error for other blocks depends on several parameters that are analyzed at this level and in particular on the microprocessor workload (i.e., the executed software).

GeFIN is a microarchitecture-level fault injector built on top of Gem5 [144], a cycle-accurate full-system simulator. GeFIN delivers very accurate results for array-based structures and it gives the opportunity to run experiments for two of the major ISAs (ARM and x86). GeFIN has two different operation mode: the IRS (Injection Runs up to the Software level) mode and the IRE (Injection Runs up to the End) mode. Concerning IRS mode, fault injection simulations end at the first visible fault effect at software layer (i.e., the moment that the first instruction affected by the fault commits to the architectural state), when faults in the hardware structures manifest as errors at the output of the CPU. These errors are then translated into the proper SFMs according to Table 6.1. For IRE, instead, injection experiments are executed up to the end of the benchmark and they are classified on the basis of the application output (i.e., masked, SDC and DUE). While IRS mode is very fast, IRE requires a lot of time but it delivers accurate results. For this reason, IRE mode is just employed to compare the reliability estimations computed by the proposed Bayesian model.

Figure 6.7 summarizes the behavior of the two GeFIN fault injector modes. The pre-fault period represents the interval from the start of the benchmark until the fault injection. This period consumes an average of about 50% of the simulation lifetime. GeFIN can skip the pre-fault period in both IRE and IRS modes, resulting in a further speedup.

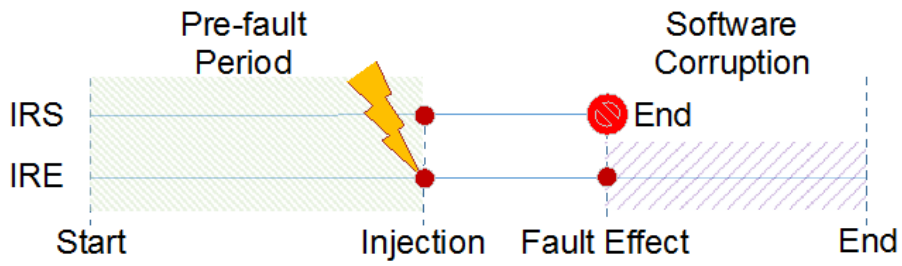


Fig. 6.7 IRE and IRS modes of GeFIN operation (Source: [4]).

6.4.3 Software Domain

The SwD models the software architecture by considering SFMs and SMs. SFMs are the error sources for this domain and their CPTs are computed as the final outcome of the HwD analysis using GeFIN. LIFILL [145], a tool developed by "Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier", was employed to characterize the CPTs of the software nodes. LIFILL is a software-level fault injector based on a virtual ISA, LLVM. In detail, software injections are performed by mutating the LLVM code according to the injected SFM. Resorting to LLVM allows to decouple the architectural level, characterized by its own ISA, from the software execution platform, thus introducing flexibility.

6.4.4 System Domain

All tools described in the previous sections were integrated into a system level reliability analyzer that implements the high-level BN model. The tool whose GUI is shown in Figure 6.8 is written in C++ and QT and provides the following functionalities to the user:

- **System architecture design**: the graph based system architecture can be easily designed. The architecture of complex hardware components (e.g., complex microprocessors) can be selected from a library of components and customized in terms of parameters, whereas the software architecture can be automatically derived from the function call graph provided by the LIFILL tool.

- **Quantitative model**: the output of the different supporting tools can be directly imported in the system analyzer to automatically compile the quantitative model of the system.
- **Reliability analysis**: once a system is described, Bayesian analysis can be performed on the resulting network. Both predictive and diagnostic reasoning is implemented and available to the user.

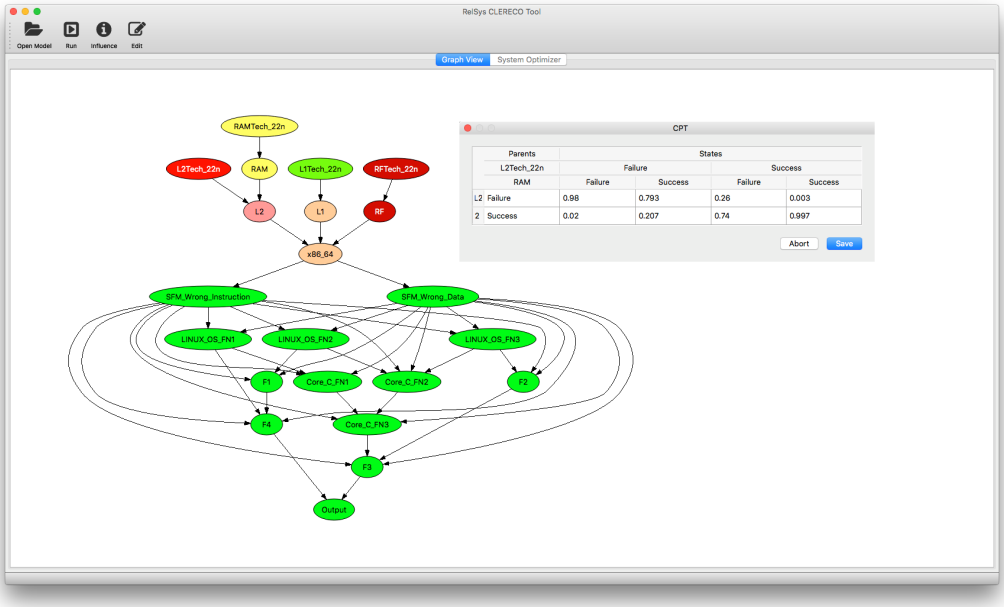


Fig. 6.8 System level reliability analyzer interface (Source: [4]).

6.5 Results

6.5.1 The experimental setup

The experimental campaign was organized into experiments on a set of carefully selected use cases. Hereinafter, each use case is referred to as System Under Evaluation (SUE). The reliability analysis targets several system configuration, taking into account different hardware architectures and application software:

- **Hardware architectures**: microprocessor based systems are one of the major classes of contemporary digital systems. We focus our experiments on this

class of products. Three relevant single core microprocessor architectures are considered (for details see Table 6.3):

  - ARM Cortex®-A9
  - ARM Cortex®-A15
  - Intel®-like i7-skylake

- **Application software**: we consider both benchmark applications and real industrial use-cases. For the benchmark applications, the software is selected from the MiBench benchmark suite [92] that has been extensively used in reliability-related literature. The selected benchmarks are:

  - String search algorithm (stringsearch)
  - Susan image smoothing algorithm (susan_s)
  - Susan image edge detection algorithm (susan_e)
  - Susan image corner detection algorithm (susan_c)
  - Rrijndael encoding algorithm performing AES encryption (aes_enc)
  - Qsort array sorting algorithm (qsort_full)
  - Fast Fourier Transform algorithm (fft)
  - Secure Hash Algorithm (sha)

  For the industrial use cases we have selected three applications from different domains:

  - FMS: it is a representative example of an avionic embedded application. This application runs on the ARM Cortex®-A15 microprocessor.
  - Motor controller (MC): it is a representative example of an industrial automation embedded application. This application runs on the ARM Cortex®-A9 microprocessor.
  - Sierpinski framework for tsunami predictor (Sierpinski): it is an open source software providing a representative example of an HPC application. This application runs on the Intel®-like i7-skylake microprocessor.

- **Operating systems**: we use both bare-metal and OS applications. For the OS applications, the Linux operating system is considered.

Table 6.3 Microprocessor architectures details.

	ARM Cortex®-A9	**ARM Cortex®-A15**	**Intel®-like i7-skylake**
Type	Out-of-Order Superscalar	Out-of-Order Superscalar	Out-of-Order Superscalar
Frequency	800 MHz - 2 GHz	1 GHz - 2 GHz	4 GHz
Technology	65/45 nm	32/28 nm	14 nm FinFET
Register File	56 32-bit registers	128 32-bit registers	168 64-bit registers
L1 I/D Caches	32 KB	32 KB	32 KB
L2 Cache	512 KB	1 MB	1 MB

6.5.2 The validation framework

The reliability analysis obtained by the proposed methodology is compared against 2 different workflows based on state-of-the-art techniques and commercial tools (Figure 6.9): the Register Transfer Reliability Analysis (RTRA) and the Microarchitectural Level Reliability Analysis (MLRA). Hereinafter, the proposed methodology is also referred to as Bayesian Reliability Analysis (BRA).

Register Transfer Reliability Analysis

Register Transfer Reliability Analysis is based on Register Transfer Level (RTL) fault injection. It is a traditional reliability analysis workflow based on a set of commercial tools. This workflow serves as a benchmark to compare our results with those that can currently be obtained using state-of-the-art commercial tools. In particular, we are interested in comparing the accuracy of our methodology in relation to the computational complexity required to perform the analysis. Faults are artificially injected into a very detailed RTL model of the hardware architecture that strictly resembles the actual implementation. This provides very accurate simulations that however need to take into account a set of drawbacks and restrictions.

RTL models of complex circuits (e.g., microprocessors, GPUs, etc.) are rarely available to system designers, especially in the early phases of the design process. This is a big obstacle that prevents the application of this reliability analysis technique in several application domains. To implement the RTRA workflow, the ARM

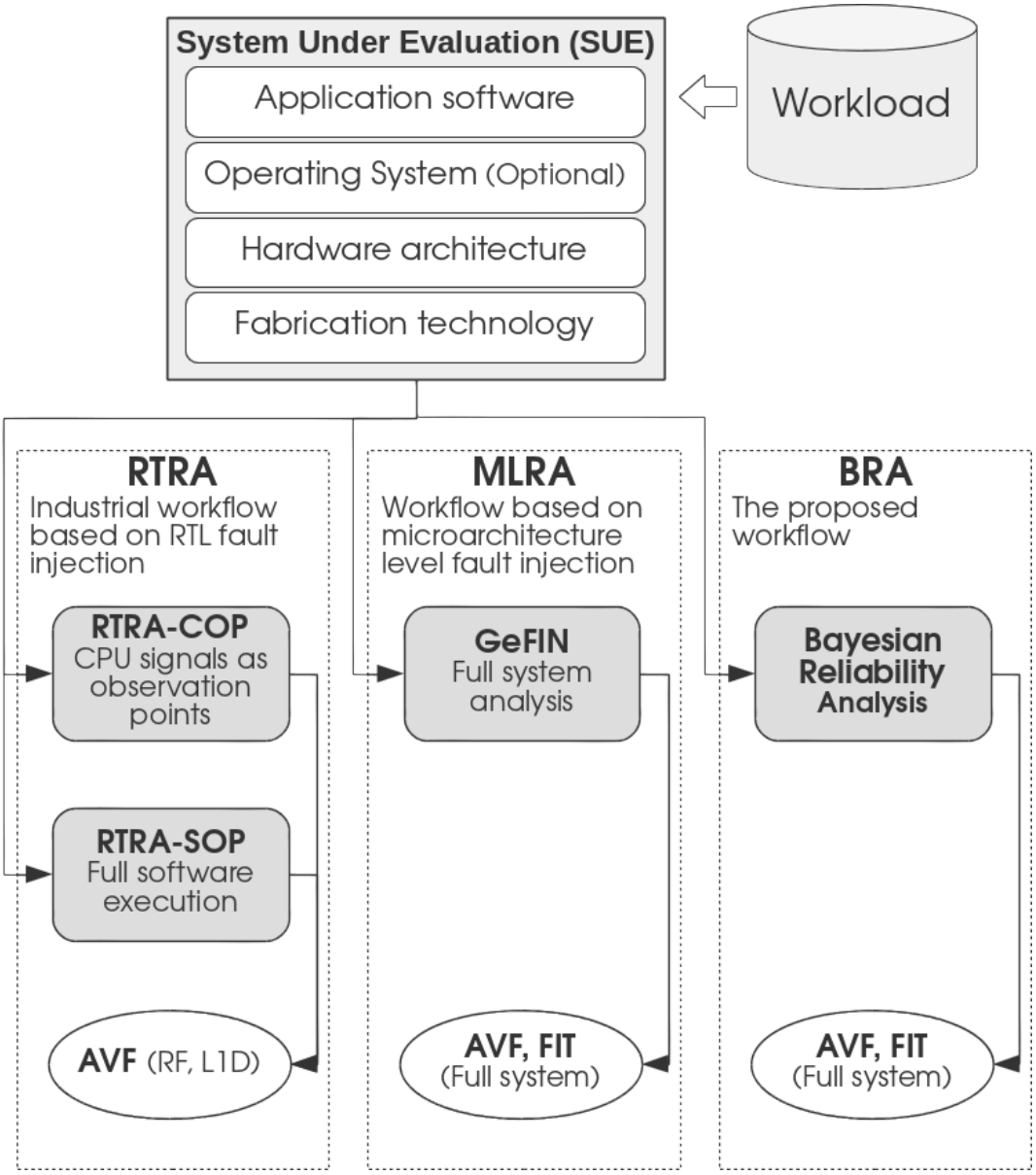


Fig. 6.9 Overview of the simulation and validation campaign.

Cortex®-A9 with the related RTL fault-injection framework were selected for the experiments since no accurate RTL models of neither an Intel®-like i7 architecture nor ARM Cortex®-A15 were available. Therefore, comparison for RTRA experiments are limited to the ARM Cortex®-A9 architecture. Even if the RTL model of the ARM Cortex®-A9 is available, other limitations must be considered. Given the complexity of the RTL model of a real microprocessor, not all internal memory arrays are fully modeled and supported by the RTL fault injection process since it

would have created serious simulation time issues (as reported later in this Chapter). The memory arrays targeted by RTRA are the L1 Data Cache and the Register File. Finally, RTRA workflow does not allow to consider applications on top of an operating system because of a huge increment of time required by the simulation. For this reason, all the applications are executed bare-metal.

Concerning the commercial tools used for the fault injection campaigns, two reliability analysis workflows are devised. They are named (1) RT Level Reliability Analysis with CPU Observation Points (RTRA-COP) and (2) RT Level Reliability Analysis with Software Observation Points (RTRA-SOP). The key difference between these two approaches is related to the fault observation points, i.e., the way the golden simulation and the faulty simulations are compared to each other.

The RTRA-COP is the standard industrial methodology used to perform safety analysis of custom embedded systems. It can be considered as the state of the art of the RTL fault injection simulations. According to this methodology, the observation points used to classify the effect of a fault are the CPU I/O pins. This means that, in order to classify the fault as dangerous or masked, the comparison between the golden simulation and the faulty simulation is performed by observing the values generated at the CPU pins (at run-time, during the simulations). The injection flow is reported in Figure 6.10 and can be summarized as follows:

- Start a new RTL simulation for each fault
- Inject the fault (bit-flip)
- The simulation runs at most for T1 clock cycles starting from the injection time. If no differences are detected between the golden simulation and the faulty simulation within T1 clock cycles (compared on a cycle-by-cycle basis), then the simulation is dropped and the fault is classified as masked. On the contrary, if there are differences between the golden simulation and the faulty simulation within T1 clock cycles, then the simulation is dropped and the fault is classified as dangerous.

The parameter T1 is estimated according to the Safety Engineer expertise, the device constrains, the test bench features, the fault list, etc. Some estimation tests can be performed in order to better estimate this parameter. However, the T1 estimation follows a very conservative approach.

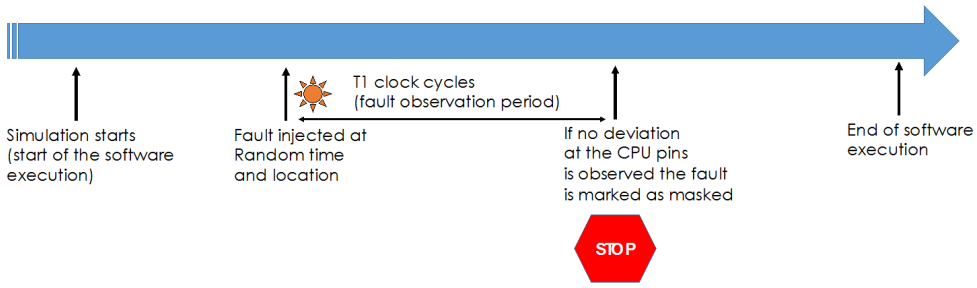


Fig. 6.10 RTRA-COP Injection timeline.

While RTRA-COP is effective to characterize how a circuit reacts to a fault, it fails to capture the masking contribution that the software executed on the architecture provides. The RTRA-SOP analysis allows for a fair comparison with the proposed methodology. In the RTRA-SOP the effect of a fault is classified looking at the result of the computation (software output) and not looking at the CPU output pins. Simulating the full software execution in a RTL simulation is a very challenging process with several performance issues that will be better discussed when presenting the experimental results.

The RTRA-SOP injection workflow is reported in Figure 6.10 and can be summarized as follows:

- Start a new RTL simulation for each fault
- Inject the fault (bit-flip)
- After the injection, if no differences are detected on the CPU pins within T1 clock cycles, then the simulation is dropped and the fault is classified as safe, otherwise the simulation is run until the end
- For each fault simulation the output is compared with the golden output. If the outcomes differ, then the fault is considered dangerous. If not, the fault is considered masked.

Similar to RTRA-COP also the RTRA-SOP resorts to the T1 parameter to avoid the full simulation of all injected faults. This is a very critical parameter that decides how many injections must be executed until the end of the software execution. Lower values enable faster injection but loose accuracy. Higher values increase the accuracy by worsen the injection throughput. The set of faults injected with this methodology

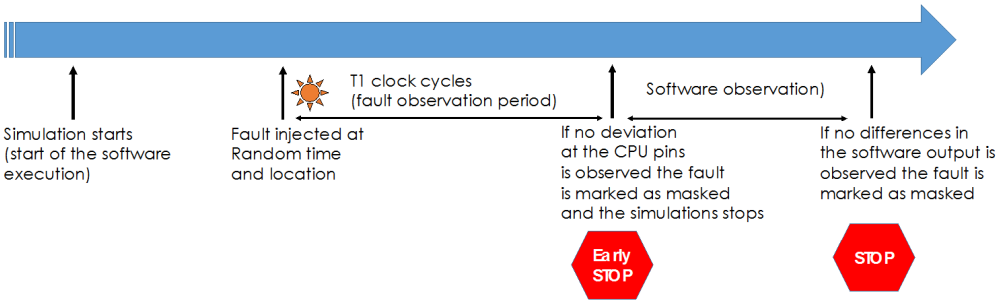


Fig. 6.11 RTRA-SOP Injection timeline.

is exactly the same injected with the RTRA-COP methodology in order to have fully comparable results.

Microarchitectural Level Reliability Analysis

Microarchitectural Level Reliability Analysis (MLRA) is based on fault injection at microarchitectural level. MLRA is chosen to overcome the limitations imposed by RTRA. MLRA allows to include complex hardware blocks (e.g., CPUs, GPUs, etc.) in the reliability analysis. Microarchitectural models are in general available for main families of components and enable fast simulations while keeping enough details to perform a precise analysis. Microarchitectural models are also easier to develop than RTL models, they can be developed by third parties and they are often released under different types of open-source licenses. For our experiments GeFIN [143] is employed. As already mentioned before, GeFIN was developed by UoA and it is built on top of Gem5 simulator. GeFIN can be configured to precisely model the microarchitecture of both ARM Cortex®-A9 and Intel®-like i7-skylake used in our experiments. It can model the behavior of several types of fault models including the soft errors considered here. Using the MLRA workflow, the AVF of all the considered memory arrays can be computed both for the ARM Cortex®-A9 and Intel®-like i7-skylake architecture. Finally combining the AVF of each component, the FIT rate can be also computed (Equation 2.4).

6.5.3 Experimental Results

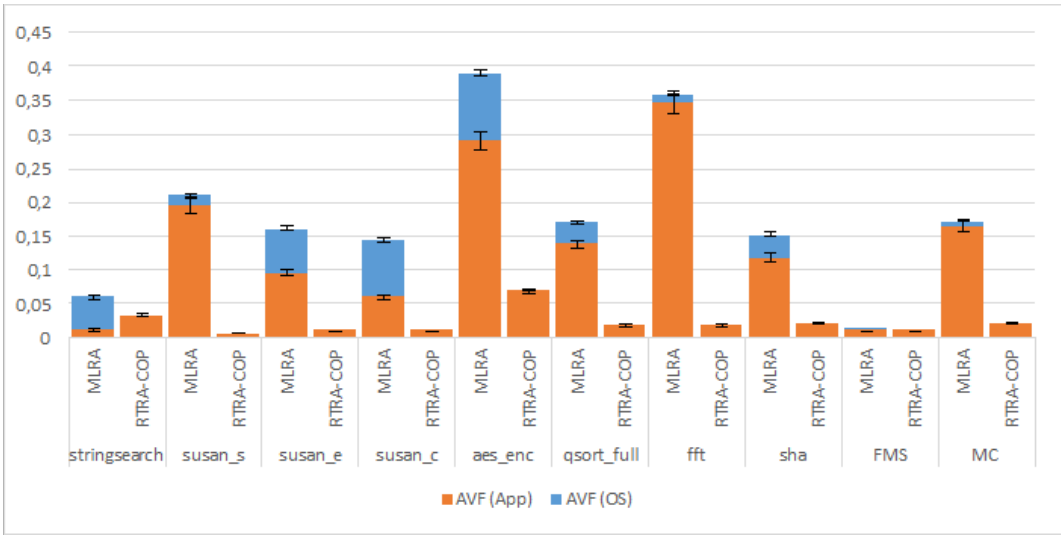
The first experiments we performed aim at understanding the accuracy of the different workflows. To give the reader a deep understanding and a complete view of the issues

related to reliability evaluation, we start comparing MLRA estimations with RTRA estimations that represent the state-of-the-art commercial reference. Later in this Section MLRA results are compared with BRA. As previously described this first comparison is limited to the observation of the effect of faults into the ARM Cortex®-A9 Register File (RF) and L1 data cache (L1D) because of the limits imposed by the RTRA. For each component of each SUE we performed 680 injections, leading to 5% error margin and 99% confidence level (see Equation 4.12). The comparison between MRLA and RTRA accuracy is performed using the AVF metric. Since MLRA takes into account the Operating System (OS), while RTRA does not, to perform a fair comparison, the AVF computed using MLRA is split into its OS and software application portion in order to focus the comparison on the application portion. This was possible by identifying for each injected fault in MLRA whether the microprocessor was in kernel or user mode.

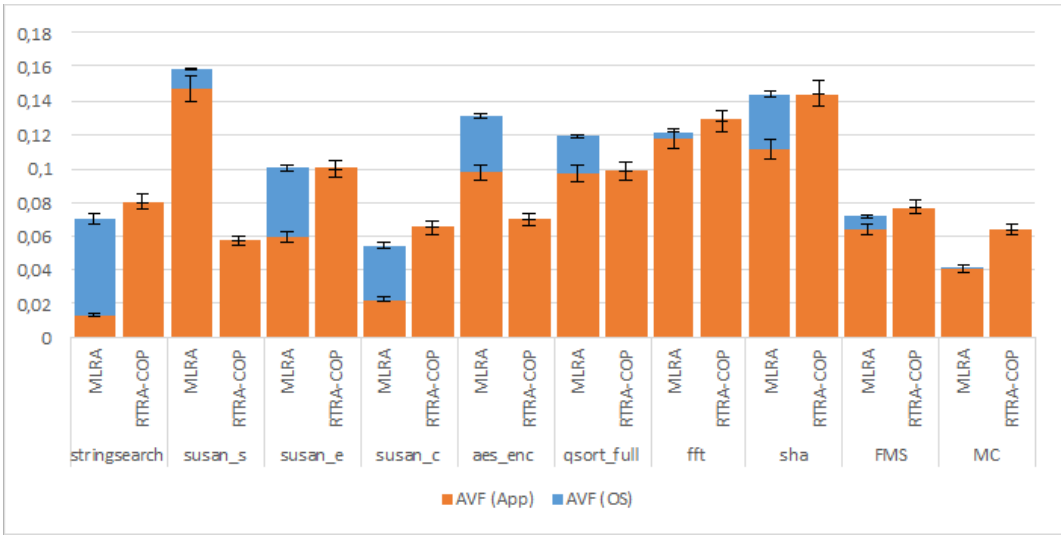
Figure 6.12 compares the AVF using the RTRA-COP workflow with the one computed using the MLRA workflow. We consider the 8 miBench applications plus FMS and MC that are designed to work with the ARM microprocessor architecture for which we can apply the RTRA-COP workflow. The execution time of Tsunami application is excessively long to run it at the RTL level.

It is important to report that in this comparison we used the ARM Cortex®-A9 as an execution platform also for the FMS application. The execution platform for FMS is therefore different from the one of the real implementation that is based on the ARM Cortex®-A15. This allows us to easily perform comparisons among different applications. Reliability of FMS on the ARM Cortex®-A15 architecture is discussed later in the section.

Interestingly, the results show a significant deviation between MLRA and RTRA-COP, with RTRA-COP providing optimistic results especially in the case L1D. However, this is not surprising. RTRA-COP monitors the behavior of the system looking at the CPU pins. While this approach is very effective when analyzing the effect on faults in the control logic of a circuit that usually manifest within a few clock cycles at the output of the CPU, it is not able to capture the logic masking effect provided by the software execution. Moreover, CPU pins (aka core pins) considered by the RTRA are positioned between the CPU the L1 caches, meaning that any deviation that does not go outside L1 in the hierarchy (i.e., L2 cache memory) is



(a) Comparison of MLRA vs. RTRA-COP AVF for L1D



(b) Comparison of MLRA vs. RTRA-COP AVF for RF

Fig. 6.12 Comparison of MLRA vs. RTRA-COP AVF estimation with 5% error margin and 99% confidence level. All benchmarks are executed on ARM Cortex®-A9 microprocessor models at the microarchitecture level (MLRA) and the RTL level (RTRA-COP).

not properly captured. To worsen the situation, the T1 timer limits the amount of time we spend to observe the effect of the fault. In our case T1 is set to 20,000 clock cycles to keep the simulation time under control. This is a reasonable value set according to good practices for the implementation of RTRA-COP. However, the duration of the full software application is some orders of magnitude higher than T1 timer. Therefore, several latent faults escape the analysis. This is particularly

evident for L1D. Data in L1D live much longer than data in RF, and therefore, the probability of a latent fault escape is higher. Differently, data in RF have a short life and the probability of latent fault escape is significantly lower than the L1D. We can therefore conclude that while RTRA-COP is a valuable instrument for a low level analysis of the circuit behavior, but it fails to take into account the effect of the software execution at the system level, thus highlighting the relevance of the need of new valuable instrument to perform this type of analysis.

MLRA and RTRA-SOP are compared to better understand the accuracy of MLRA. In facts, RTRA-SOP mimics the way MLRA performs the reliability analysis. However, a serious performance issue arises when implementing RTRA-SOP. Figure 6.13 shows the days of simulation that would be required to perform the RTRA-SOP analysis injecting 680 faults for RF and 680 faults for L1D. Performance is reported with a setup composed of a single workstation (Intel® Xeon based with 64GB of RAM) performing injection sequentially (for a single thread injector).

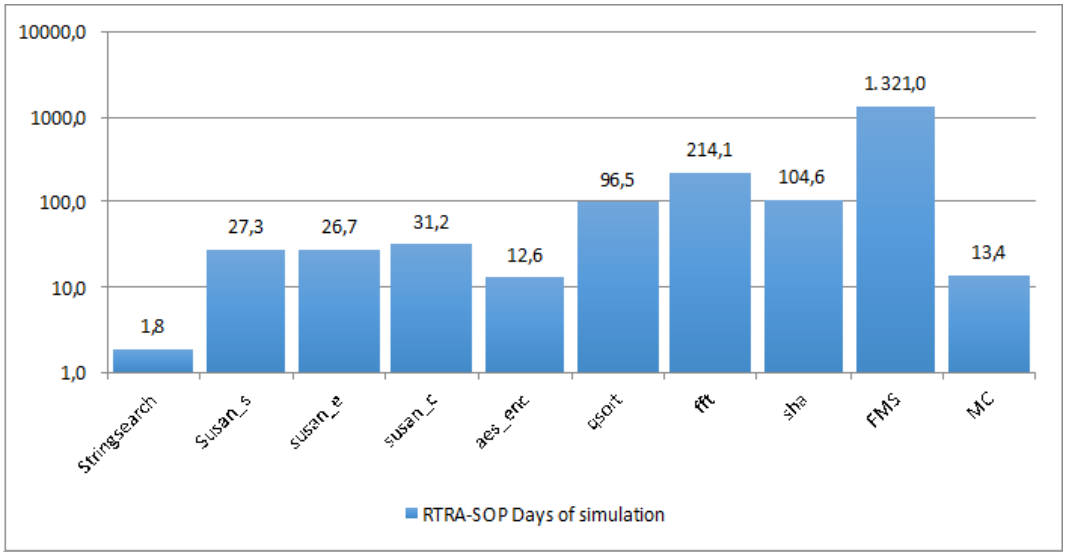


Fig. 6.13 CPU time in days of simulation to perform 680 injections for L1D and 680 injections for RF using RTRA-SOP with T1 timer equal to the duration of the program. Simulation time is provided in Days of simulation using a logarithmic scale.

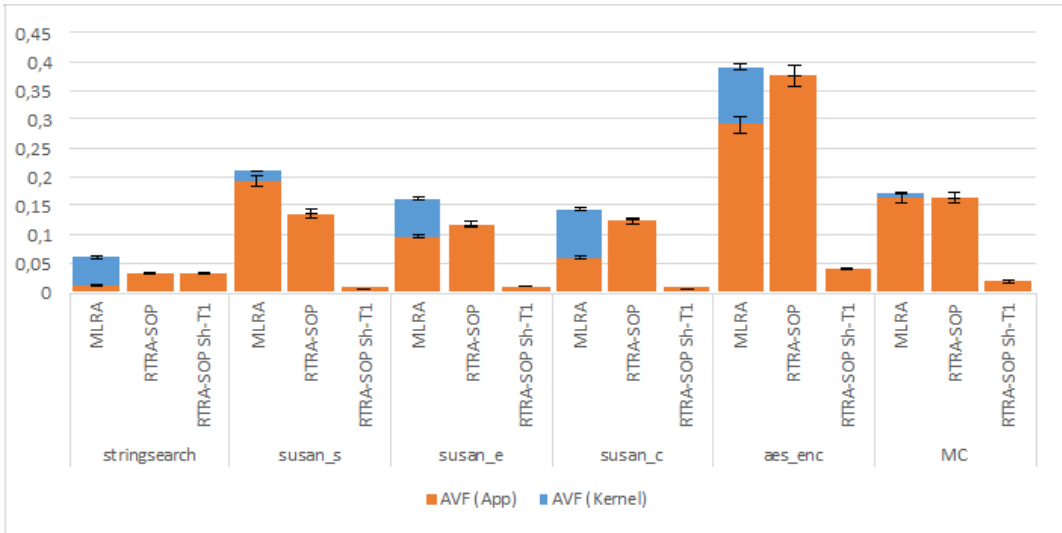
It is clear that, even with the use of multiple threads on the same machine and multiple workstations to parallelize the simulations, not all applications can be analyzed with this technique in a reasonable time. We therefore limited this analysis

to stringsearch, susan_s, aes_enc, susan_e, susan_c and MC whose simulation time was reasonable to handle with the available resources.

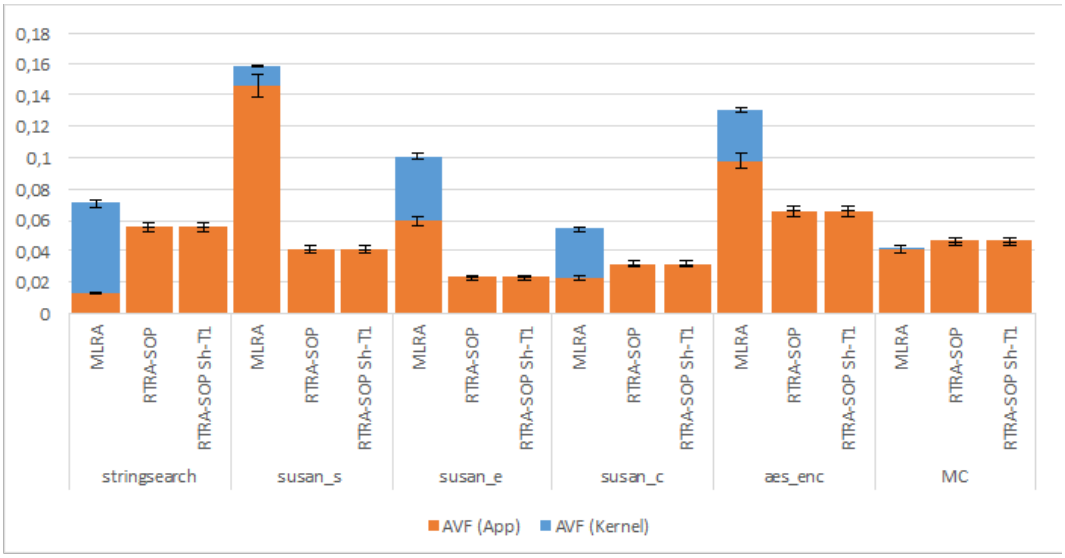
Figure 6.14 compares the AVF using the RTRA-SOP workflow with the one computed using the MLRA workflow for the selected SUEs. For RTRA-SOP we performed two simulation campaigns. The first uses the time T1 set to 20,000 clock cycles exactly as for the RTRA-COP workflow. The second simulates all injected faults until the end. As it is evident looking at Figure 6.14, the timer T1 is a critical parameter also for RTRA-SOP. Choosing a low T1 leads to very significant deviations in the estimated AVF, especially considering faults in the L1D. The motivations for these deviations are exactly the same that apply for RTRA-COP. Data in L1D tend to have a lifetime longer than T1 and therefore several escapes appear. On the other hand, if T1 is set to the duration of the full application, deviations are strongly reduced and we can appreciate how MLRA is able to provide very accurate results (comparison of the orange bars that exclude the contribution of the OS). Looking at Figure 6.14, it is also very important to highlight how the OS may represent a significant influencing factor for the AVF of a system. The impossibility of RTRA of fully taking into account the influence of the OS is a main limitation of this reliability estimation.

Some considerations are required before comparing the accuracy of AVF estimations performed using MLRA and BRA. First, both MLRA and BRA are not limited to injection in the register file and in the L1 Data Cache. Therefore all major memory arrays of the microprocessor that include also the L1 Instruction cache (L1I), the L2 Cache (L2), and the Store Queue (SQ) are considered. Secondly, the ARM Cortex®-A9 is not any more the only microprocessor architecture that can be analyzed. The Intel®-like i7-skylake architecture used by the Sierpinski application is considered by this comparison too. Finally, BRA is mainly devoted to perform very early and fast design exploration rather than providing accurate reliability estimation. Table 6.4 reports the hardware configuration of the three microprocessors considered in this analysis and in particular the size of the considered memory arrays expressed in bits. In the comparison between MLRA and BRA the effects of the OS are also considered.

Figure 6.15 compares the full system AVF computed using MLRA and BRA for the 8 considered miBench benchmarks and for the three considered industrial



(a) Comparison of MLRA vs. RTRA-SOP AVF for L1D



(b) Comparison of MLRA vs. RTRA-SOP AVF for RF

Fig. 6.14 Comparison of MLRA vs RTRA-SOP AVF esitmation. All benchmarks are executed on the ARM Cortex®-A9 microprocessor RTRA-SOP and RTRA-SOP sh-T1. In RTRA-SOP the T1 time is set to the duration of the application, therefore all faults are simulated until the end of the program execution. In RTRA-SOP sh-T1 only faults that create differences at the CPU outputs in T1 clock cycles are simulated until the end of the application.

use cases. All benchmarks except Sierpinski are analyzed on the ARM Cortex®-A9 architecture while Sierpinski is analyzed on the Intel®-like i7-skylake architecture only.

Table 6.4 Hardware configuration for the three considered microprocessor architectures. It reports the size (#bits) of the main arrays considered in the analysis.

	ARM Cortex®-A9	**ARM Cortex®-A15**	**Intel®-like i7-skylake**
L1D	262144	262144	262144
L1I	262144	262144	262144
L2	4194304	8388608	2097152
RF	1792	4096	10752
SQ	256	512	4608

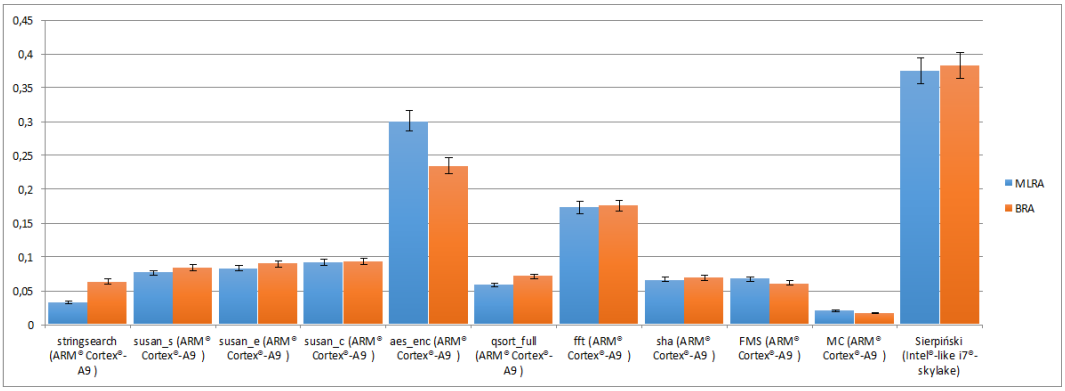


Fig. 6.15 AVF estimation comparison between MLRA and BRA. This is a full system AVF including contribution of RF, L1D, L1I, L2 and SQ.

It is very interesting to discuss the differences between the two estimations that actually set the accuracy of BRA vs. MLRA. Even if BRA is not designed for very high accuracy, deviations in the AVF estimations are very limited even for the most complex use cases. The only outlier in these experiments is the AES benchmark, which is a peculiar software architecture storing several constant data structures in memory that require special attention when performing the software characterization. However, if we look at the absolute error, it is limited to about 7 percentile units. While this deviation may be considered significant in the late stages of the design, when very accurate reliability assessment is required to qualify the safety of the developed system, it is more than acceptable in the early phases of the design when reliability assessment is required as a reference metric to understand how to optimize the system to reach the desired reliability level without incurring in worst-case design.

To prove the flexibility of the proposed methodology, we also performed the reliability evaluation for FMS for ARM Cortex®-A15. Figure 6.16 shows the

comparison of the two architectures (ARM Cortex®-A9 and ARM Cortex®-A15) reporting AVF for each component of the system. The last column shows the system AVF. BRA allows to capture the difference in terms of AVF when the same application is executed on the two different architectures, thus highlighting that it can be employed to explore the system design effectively. In this case, we observe about 4% difference. The system AVF provides a global picture of the system weighting the contribution of each block according to its size. The other bars, instead, allow to have a deepest look at the AVF of the single blocks. The deviation of the system AVF between the two architectures can be attributed to two aspects. The former is the AVF deviation of the L2 cache, the latter is the different size of this block.

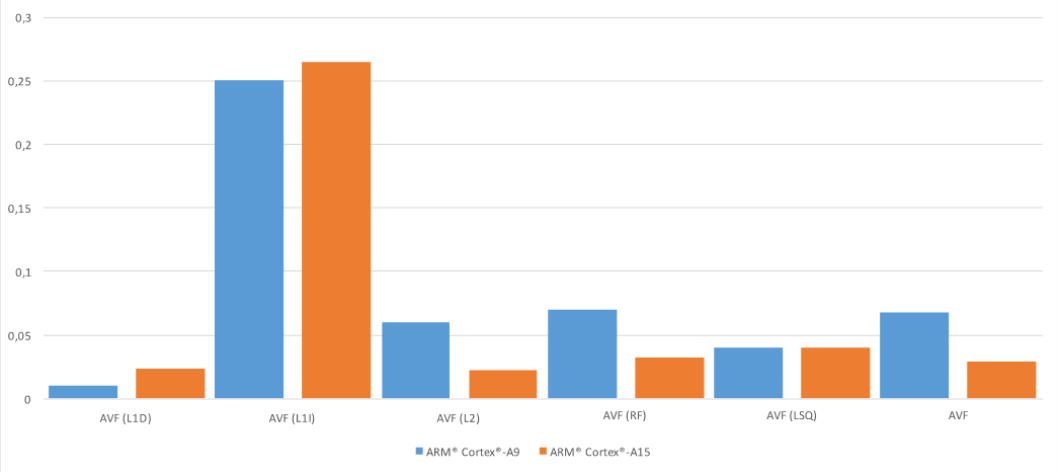


Fig. 6.16 Comparison of the AVF for two different ARM architectures running FMS: ARM Cortex®-A9 and ARM Cortex®-A15.

The time required to perform the simulations and to compute the reliability metrics is an important aspect to evaluate the scalability of the analysis with respect to the complexity of the target systems. Figure 6.13 shows that a lot of time is spent when resorting to RTRA-SOP, so much time that for some of the considered target systems this analysis is not feasible. On average, MLRA is more than one order of magnitude faster than RTRA-SOP.

Figure 6.17 introduces a comparison between MLRA and BRA in terms of time required by the reliability analysis considering all the hardware blocks of the systems. Results are reported in terms of hours of computation. In most of the cases BRA computation time is significantly reduced with respect to MLRA. This means that

the reduced accuracy of this technique is compensated by a fast computation time, that is a valuable aspect in the design phase to reduce time-to-market.

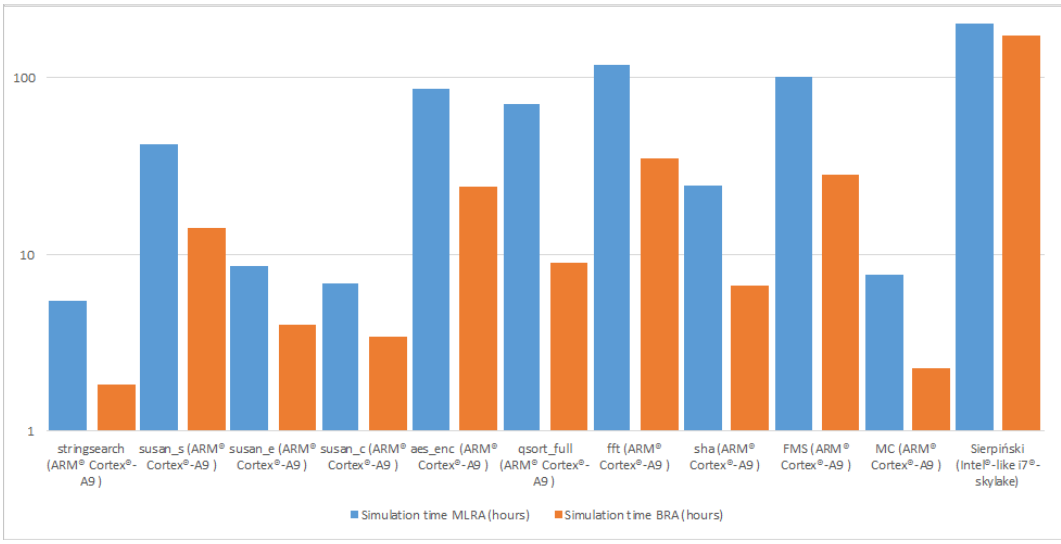


Fig. 6.17 Simulation time comparison between MLRA and BRA. Simulation time is provided in hours of simulation using a logarithmic scale.

How the total simulation time for BRA is distributed among the characterization of the three system levels (hardware architecture, software and system) is shown in Figure 6.18. The main complexity of this analysis is the hardware characterization. Moreover, once the Bayesian model of a system has been constructed, the time required for its analysis and the time required to perform statistical reasoning on the model is negligible, thus demonstrating that it has the potential to be a valuable instrument for the system designer.

To conclude the discussion regarding the complexity of the proposed methodology, it is important to have a look at the complexity of the Bayesian model that is the base for this analysis. Table 6.5 gives a clear indication of this complexity. It reports the number of nodes of the model (both real components of the system or meta-nodes required to properly model the information propagation through the network) and information on the In-degree of each node. Looking at the table it is important to highlight that regardless of the complexity of the model that ranges from a few tens of nodes to more than a thousand of nodes the simulation time is constant, because the size of the CPTs remains small. This result highlights the effectiveness of the solution introduced in Subsection 6.3.2, which limits the in- degree of software

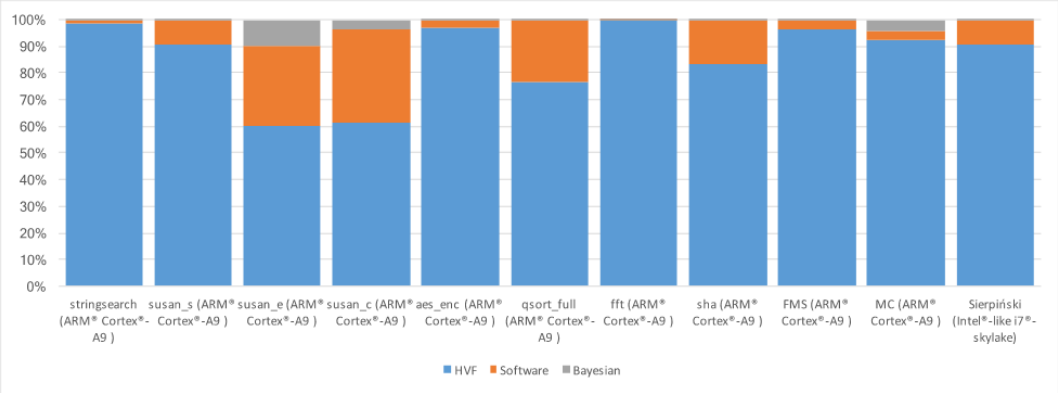


Fig. 6.18 Complexity distribution between the different layer.

nodes. While this technique slightly increases the number of nodes of the network, it allows to take the complexity of the analysis under control since the size of the CPTs depends on the number of incoming edges of each node.

Table 6.5 Complexity of the Bayesian Model required for BRA.

	Node Count	**Average in-degree**	**Max in-degree**
Stringsearch	180	1.744	8
susan_s	336	1.78	8
susan_e	376	1.782	8
susan_c	337	1.777	8
aes_enc	256	1.773	8
qsort_full	142	1.718	8
fft	297	1.774	8
sha	333	1.805	8
FMS	1560	1.927	8
MC	55	1.655	8
Sierpinski	1031	1.99	8

To give a complete insight into the reliability of the analyzed systems we computed other reliability metrics: the FIT (see Chapter 2) and the EPF (see Chapter 4).

In order to compute the FIT rate, λ_S, (the FIT is computed similarly to to Equation 4.3), we need to identify the FIT rate of the target technology, λ. This task is managed by the tool presented in Subsection 6.4.1. Figure 6.19 reports the FIT rate per bit

of 6T SRAMS for the technology nodes used in the fabrication of the considered microprocessor architectures:

- 14 nm Bulk FinFET used by the Intel®-like i7-skylake
- 65/40 nm Bulk Planar used by the ARM Cortex®-A9
- 32/28 nm Bulk Planar used by the ARM Cortex®-A15

The figure shows that the target technology has a significant impact on the FIT rate of a system. However, it is also one of the aspects of a design that has very reduced degree of freedom for the system designer. The provided FIT rates are for a single 6T SRAM cells operating in typical conditions (1V, 50C).

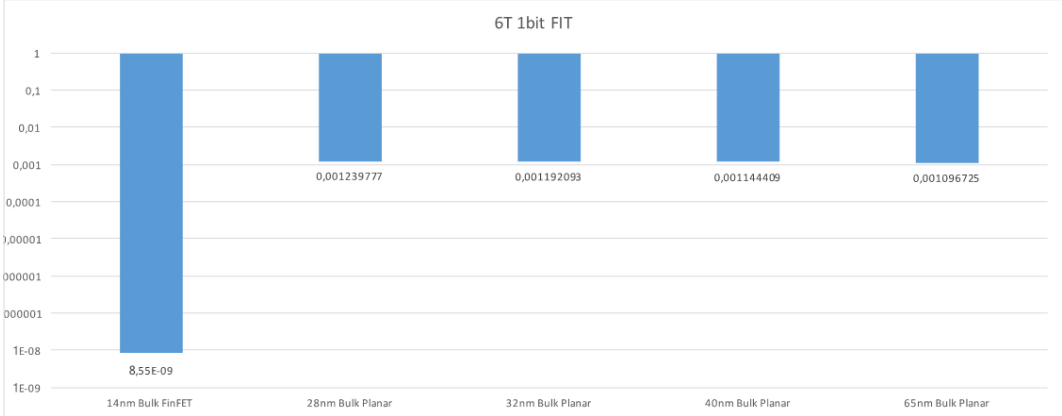


Fig. 6.19 6T SRAM FIT/bit for the five fabrication technologies used in the considered microprocessor architectures.

Figure 6.20 illustrates the FIT rate computed using MLRA and BRA for all systems running on the ARM Cortex®-A9. In this figure we investigated the difference between the two technology nodes commercially available for this microprocessor. The figure shows that the technology node is actually not really influential on the definition of the FIT of the system, and this is somehow expected given that the raw error rate per bit of the two technologies is very similar. On the other hand, there is a significant variance of the FIT at the system level depending on the executed application. In detail, aes_enc features the highest FIT, while the FIT reported for MC is very low, suggesting that this application is already highly resilient to soft errors and the introduction of invasive fault tolerance mechanisms might not be required.

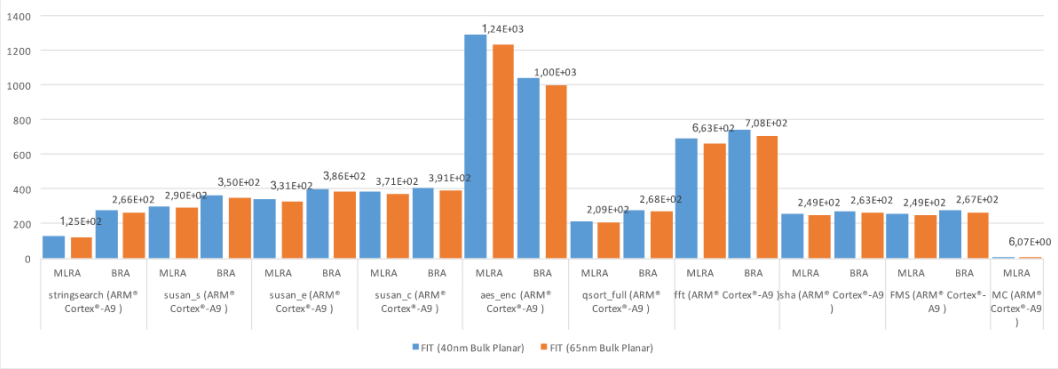


Fig. 6.20 FIT rate for the applications executed on the ARM®Cortex®-A9 under two different technology nodes.

Figure 6.21 reports the FIT rate for the Sierpinski application, executed on the Intel®-like i7-skylake featuring a 14nm Bulk FinFET technology node. By comparing the FIT rate of this system with the one of the applications executed on the ARM Cortex®-A9, the FIT rate is very low. More specifically, the FinFET technology is playing a major role reducing the effect of soft errors on this system.

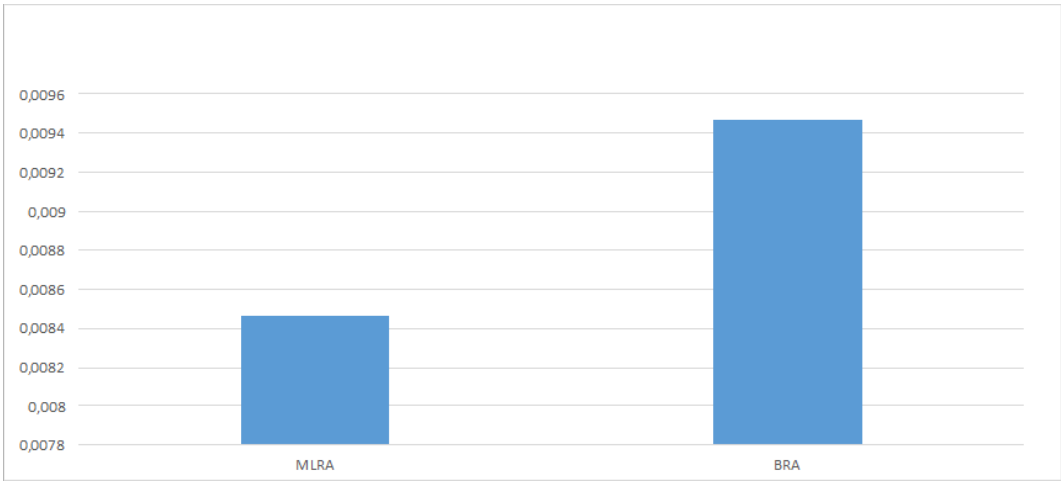


Fig. 6.21 FIT rate for the Sierpinski application executed on the Intel®-like i7-skylake 14nm Bulk FinFET technology node.

Finally, it is interesting to compare the FIT rate of the FMS application when executed on two different microprocessors architectures featuring different technologies (Figure 6.22). The difference in terms of FIT is decreased with respect to the difference of AVF for the two architectures (Figure 6.16), nevertheless that the technology nodes have quite similar raw FIT. The reduction is due to the bigger size

of the memory arrays of the ARM Cortex®-A15 (*#bit* of Equation 4.3). This is an interesting example highlighting how, from the one hand changing the architecture gives benefits in terms of AVF but on the other hand increases the number of vulnerable resources. Thanks to the proposed methodology the best trade-off can be investigated to take the most appropriate design decisions.

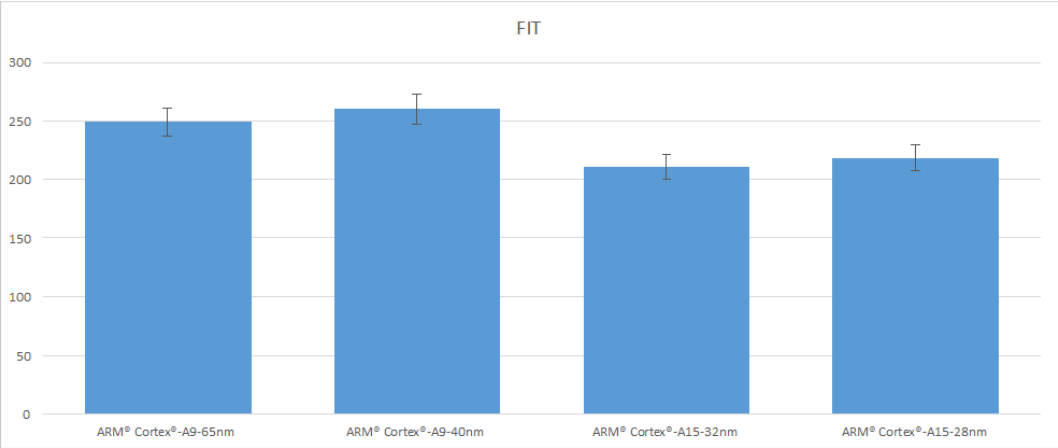


Fig. 6.22 Comparison of the FIT rate for FMS considering different microprocessor architectures fabricated using different technology nodes.

EPF results are plotted in Figure 6.23 with logarithmic scale. The comparison is performed considering the FMS application and the MC application under two implementations: (1) ARM Cortex®-A9 65nm Bulk Planar CMOS clocked at 800-MHz and (2) ARM Cortex®-A15 28nm Bulk Planar CMOS clocked at 2.5GHz. While MC benefits of a very significant increase of the EPF (one order of magnitude), the EPF of FMS is almost constant. To explain this behavior, we need to analyze the behavior of the two applications. FMS is a real-time application. Therefore, the number of times it is executed during a given time period is independent on the target architecture. As a consequence, even in case of a faster processor, it remains constant. This means that the EPF is not influenced by the performance of the microprocessor and the only benefit obtained moving from the ARM Cortex®-A9 to the ARM Cortex®-A15 is the lower FIT of the system as reported in Figure 6.20. Differently, MC benefits from the higher performance of the ARM Cortex®-A15 and therefore moving from one pplication to the other has a significant impact on the EPF.

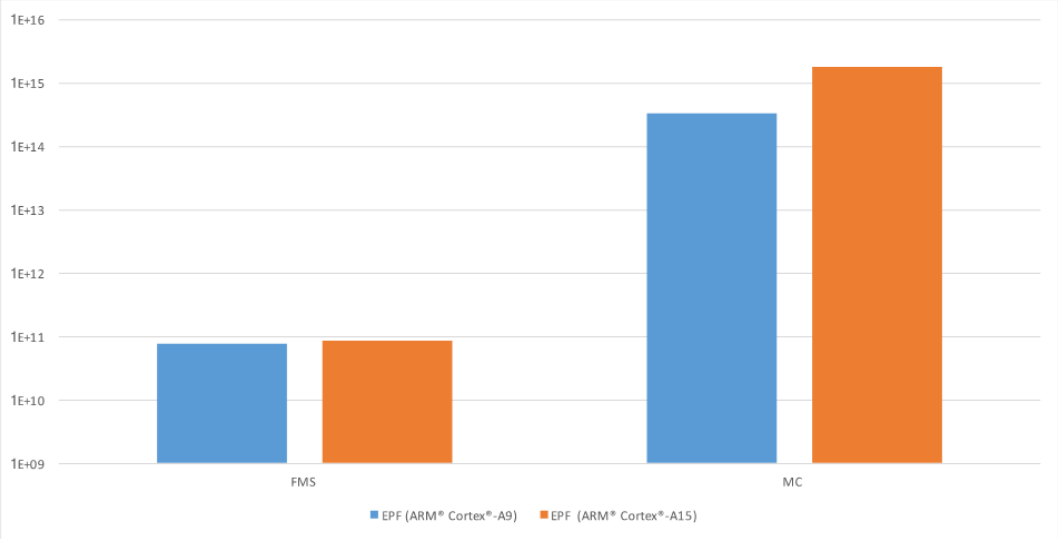
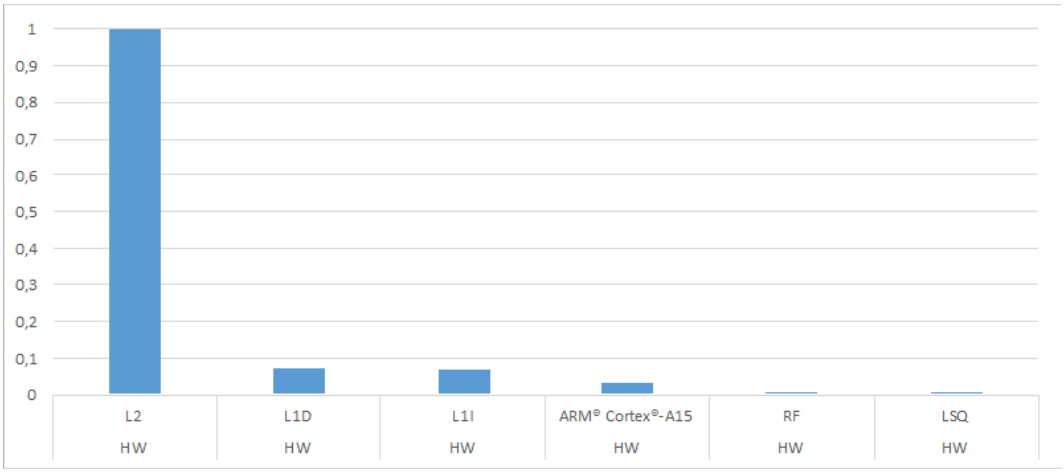


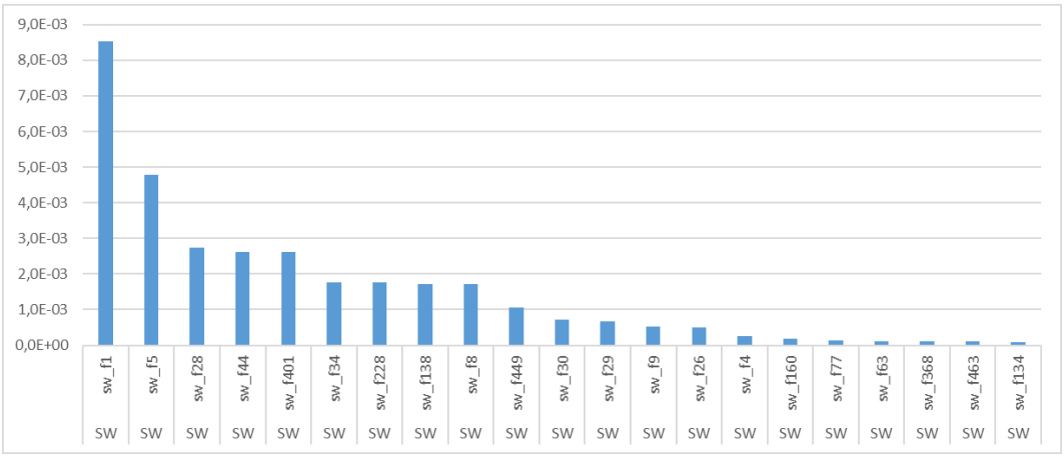
Fig. 6.23 EPF computed for FMS and MC on two different ARM architectures: ARM Cortex®-A9 implemented with 65nm Bulk Planar CMOS clocked at 800-MHz and ARM Cortex®-A15 implemented with 28nm Bulk Planar CMOS clocked at 2.5GHz. Results are plotted in a logarithmic scale.

Presented results show the great accuracy and flexibility of the proposed Bayesian Reliability Analysis when reliability must be evaluated. As anticipated, the most valuable advantage introduced by the Bayesian model is the component sensitivity analysis driven by Bayesian diagnostic reasoning, allowing the identification of the most reliability-critical components of the system. The analysis starts by inserting in the statistical model the evidence that the system is in a faulty state. Setting a Bayesian evidence means performing a hypothesis of the state of the system. Based on this hypothesis the Bayesian model allows us to update our beliefs on the state of each hardware or software component of the system. In simple terms, we can answer the following question: "if we know that the system is faulty which is the state of each component of the system?". This somehow allows us to identify where the faults resulting as errors manifest. Scores of every system components are normalized to give an indication of their sensitivity to faults. Figure 6.24 shows the result of the sensitivity analysis performed for FMS.

Results of the sensitivity analysis must be carefully analyzed to avoid wrong conclusions. First of all, since the Bayesian model decouples the different layers of the system, sensitivity scores must be analyzed layer by layer. More specifically, the



(a) Sensitivity analysis for hardware



(b) Sensitivity analysis for software

Fig. 6.24 Sensitivity analysis performed for FMS running on the ARM Cortex®-A15.

sensitivity score is a normalized score and its absolute value is not significant but it is useful to compare different components.

Concerning the hardware domain in Figure 6.24, the result is somehow expected for this type of system. In detail, the hardware domain is a low level layer and therefore its sensitivity is highly influenced by the raw technological error rate. If a fault strikes the system, it is very likely to manifest in the L2 cache since it exhibits the largest number of bits. However, sensitivity scores must not be analyzed alone. Their power arises when they are analyzed together with the knowledge of the architecture of the system. In particular, Figure 6.25 provides a snapshot of the Bayesian model for the hardware architecture of the ARM Cortex®-A15. L2 cache

is hierarchically connected to L1I and L1D that have a significant lower sensitivity score. This means that when moving from one block to another block in the hierarchy faults may disappear. This type of analysis is very valuable. The designer can exploit it to take proper decisions on where to insert fault tolerance mechanisms.

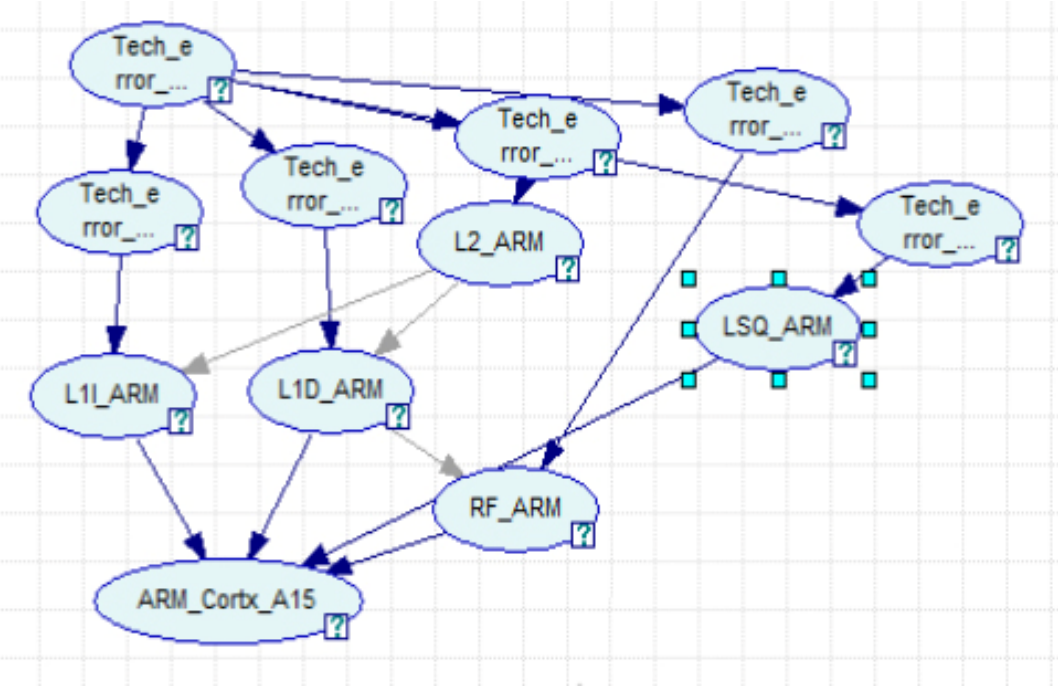


Fig. 6.25 Snapshot of Bayesian model of the hardware portion of the ARM Cortex®-A15.

However, a problem arises when the complexity of the architecture increases. This is for example the case when we consider the software domain. Figure 6.26 shows a snapshot of the full FMS Bayesian reliability model including all nodes modeling the software layer. It is clear that the complexity of this model is too high and a global visual inspection of the hierarchy is not feasible unless the designer focuses on some specific portions of the system. However, while human inspection of the sensitivity score coupled with the hierarchy of the system is hard, automatic algorithms can perform this analysis and suggest optimizations for the system.

An automatic system optimizer built on top of the proposed Bayesian model is described later in Chapter 7. The main goal of the results presented in this chapter is to highlight that the proposed methodology is well suited to be further employed for system optimization, thus introducing a valuable improvement with respect to state-of-the-art techniques.

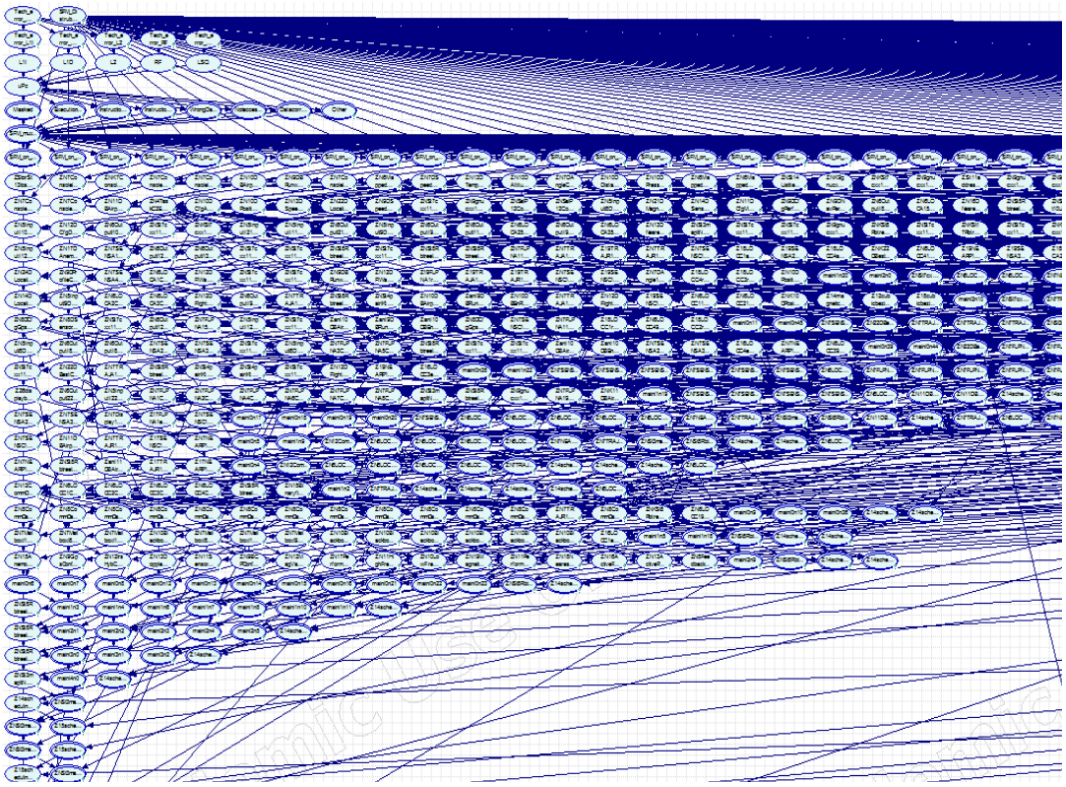


Fig. 6.26 Snapshot of the full FMS Bayesian model.

6.6 Conclusions

In this chapter we proposed a scalable, cross-layer methodology and supporting tools ecosystem for accurate and fast estimations of computing systems reliability based on a component-based Bayesian network model. The model and related tools address all layers of a complex system from the technology up to the application software. The proposed methodology is compared with state-of-the-art and industrial reliability analysis workflows. We showed that it provides accurate reliability estimations (in the worst case AVF absolute error with respect to MLRA is equal to 7 percentile units). Thanks to its flexibility many complex systems can be analyzed even at early stages of the design, varying technology, hardware microarchitecture, software application, and OS configuration. Time required to perform the proposed reliability analysis is significantly reduced with respect to MLRA (several orders of magnitude with respect to RTRA). Besides the results concerning reliability assessments, one of the key capabilities of the proposed framework is the possibility to perform early diagnostic analysis to identify reliability-critical components of the system, thus

enabling quick design exploration by evaluating the effects that different cross-layer protection mechanisms at the technology, hardware and software layers. The design exploration and the related system optimization are discussed in the next chapter.

Chapter 7

Reliability optimization of complex digital systems

Designing reliable systems is a complex engineering task that nowadays follows a cross-layer approach, requiring a careful planning for different fault-tolerance mechanisms and design solutions to be applied at different system's layers, starting from the technology up to the software domain. While these design decisions have a positive effect on the reliability of the system, they usually have a detrimental effect on its size, power consumption, performance and cost. The optimization of a system for cross-layer reliability is therefore a multi-objective optimization problem in which reliability must be traded-off with other design dimensions. Tools to support the automatic optimization of reliable systems fully exploiting the potential of cross-layer reliability solutions and able to trade-off reliability with other design dimensions are still not mature.

Chapter 6 introduced a Bayesian model able to identify the weak components of the system in terms of reliability. This chapter proposes a cross-layer multi-objective optimization algorithm for complex electronic systems based on that system-level Bayesian reliability model. The full system stack, starting from the fabrication technology, up to the software layer, is modeled and is taken into account when analyzing the reliability of the system and when making design decisions to select the best protection mechanisms to apply. A new heuristic based on the extremal optimization theory is used to efficiently explore the design space. An extended set of simulations shows the capability of this framework when applied both to

the benchmark applications and realistic systems employed in the experiments of Chapter 6 (Section 6.5).

7.1 Introduction

The optimization of a system for cross-layer reliability is a multi-objective optimization problem in which reliability must be traded-off with other design dimensions [146]. The number of constraints to consider when performing this task is rapidly growing to a level that cannot be handled any more by system designers without the support of proper automatic optimization tools.

In the reliability domain, very few publications propose automatic system level design optimization algorithms. Coit at al. [146] and Xing et al. [147] review a set of optimization techniques for the redundancy allocation problem. Most of the analyzed solutions are based on genetic algorithms and all start from the assumption that data redundancy is the only available fault-tolerance mechanism. The intrinsic resiliency of the system to selected hardware faults is not taken into account during the optimization process that only focuses on optimizing the amount of redundancy with respect to area constraints. One of the few attempts to optimize fault-tolerance mechanisms considering both the hardware and the software layer is proposed by Wattanapongsakorn and Levitan in [148]. They use simulated annealing to evaluate random configurations of all available components selecting at each iteration the best identified combination. The considered cost function is simply the sum of the cost of each component and the optimization ends when the best solution does not improve for a pre-defined number of iterations. Although the technique is interesting, the exploited reliability model is very simple (i.e., a single failure probability for each component). It therefore does not take into account the effect of the interaction of the different components on the reliability of the full system. Moreover, the trade-off between reliability and other design dimensions is not considered.

Differently from other works, Shafique et al. propose a reliability optimization framework where the sole software layer is modified during the optimization [149]. The proposed approach introduces step-by-step protection mechanisms to certain instructions until a desired level of protection is achieved or, at worst, all unprotected instructions are protected. The hardware is only considered as a source of errors that propagate to the software, and is therefore not optimized.

To the best of our knowledge, optimization strategies for the design of reliable systems fully exploiting the potential of cross-layer reliability solutions and able to trade-off reliability with other design dimensions are still missing.

This work makes a step forward to cover this gap. The proposed optimization framework is based on an extended version of the Bayesian reliability model proposed in Chapter 6. The full system stack starting from the fabrication technology, up to the software layer is modeled and is taken into account when analyzing the reliability of the system and when making design decisions to select the best protection mechanisms to apply. A new heuristic based on the extremal optimization theory [150] is used to efficiently explore the design space. Apart for being specifically designed for cross-layer reliability, the proposed heuristic is also designed for multi-objective optimization. Therefore, reliability can be efficiently traded-off with other design constraints (e.g., size, performance and power consumption).

A large campaign of experiments is reported to demonstrate the capability of the proposed framework. Experiments aim at the optimization of a set of systems based on realistic microprocessor models running a set of benchmark and real applications (analyzed systems were selected from the ones presented in Section 6.5). A large library of protection mechanisms at different layers taken from the literature is used to generate a large set of design options.

The remaining of this chapter is organized as follows: Section 7.2 presents the formalism used to model the target system while Section 7.3 overviews the optimization strategy. Section 7.4 reports and discusses the results of the performed experimental campaign and Section 7.5 summarize the main contributions and concludes the chapter.

7.2 System level modeling

In this work, systems are modeled using an extended version of the Bayesian model introduced in Chapter 6. Here the Bayesian model is briefly summarized in order to introduce the formalism adopted in the remainder of this chapter.

Figure 7.1-A shows an example of system modeling. The system denoted with S is modeled using an extended Bayesian network, i.e., a directed acyclic graph

defined as:

$$S = (N, E, \Theta, P) \tag{7.1}$$

where:

- $N = \{n_1, n_2, \cdots, n_m\}$ is the set of network nodes, each node identifying a software/hardware component of the system as will be better described later in this section. Each node is associated to a set states that in our case identify error or error-free conditions for the node (e.g., the L1 cache can be error free, it can be affected by a single bit-flip or by a double bit-flip).
- $E = \{(n_i, n_j) \in N \times N\}$ is the set of arcs that define temporal or physical relations among components, e.g., a failure state of a component may influence the state of other components.
- $\Theta = \{\theta_1, \theta_2, \cdots, \theta_m\}$ is a set of Conditional Probability Tables (CPT), each table associated to a node. The CPT θ_i of node n_i defines the probability of n_i to be in a given state, conditioned on the state of its parent nodes.
- $P = \{p_1, p_2, \cdots, p_m\}$ is a set of optional parameter tables that can be associated to each node of the network. Parameters contained in this table are used to characterize the component (e.g., area, power consumption, etc.). Any parameter that can influence the design decisions can be included in this table as far as it can be measured for each component.

As described in Chapter 6, the Bayesian model is organized in four domains: the technology domain (TD), the hardware domain (HwD), the software domain (Swd) and the system domain (SD).

When performing design optimization, the basic assumption is the availability of different implementations of the system components to form a *Component Library* (CL) as reported in Figure 7.1-B. Each implementation of a component in this library must be fully characterized with its CPT and parameters.

To support the optimization process, components of the system (i.e., nodes) are organized into a k-levels hierarchy of *clusters* following the hierarchical architecture of the real system.

As an example, Figure 7.1-A defines a 2-level hierarchy that includes at the first level two clusters: the first labeled as `uPC Cluster` grouping all nodes modeling

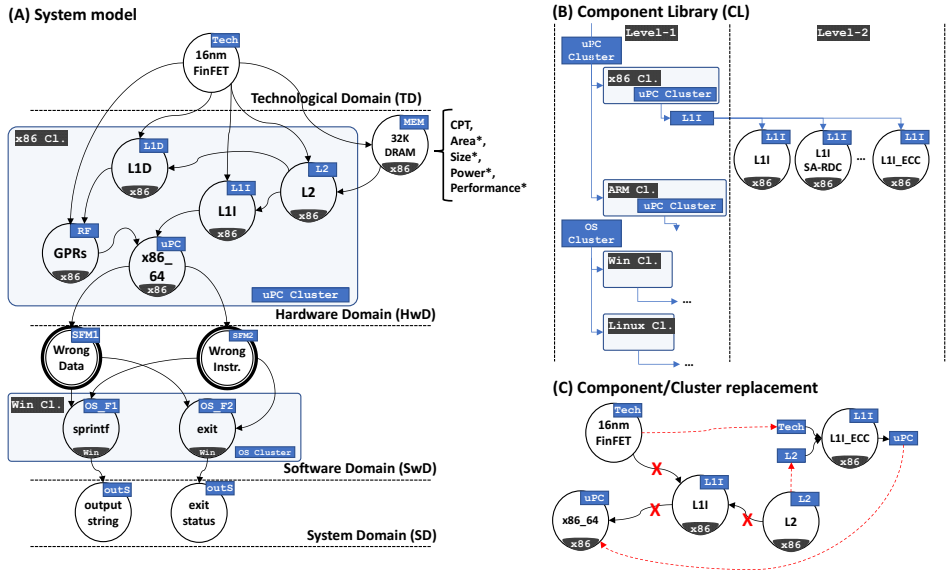


Fig. 7.1 System modeling. (A) *Bayesian model of the systems*: nodes are organized into domains and each node is characterized by a Conditional Probability Table (CP) plus a set of optional parameters (e.g., are, size, power, performance) that can be used during the optimization process, (B) *design alternatives*: for each component (node) or cluster of components different implementations are defined, thus forming a library of design alternatives, (C) *component replacement* showing how a component can be replaced with a different implementation in the model.

the components of the microprocessor, the second named `OS Cluster` grouping all nodes modeling the operating system functions and modules used by the application software. Different implementations of each cluster are available to the designer as reported in Figure 7.1-B and must be evaluated during the optimization process. Using the concept of clusters, multiple levels of hierarchy can be defined. At the end of this hierarchy, different implementations of single components are defined in the library. For example, Figure 7.1-B defines three implementations of the L1 instruction cache in the x86 cluster: the basic implementation not supporting any fault tolerance mechanism and two implementations supporting different fault tolerance mechanisms.

Using the proposed system model and the related library of components, the optimization of a system becomes an exploration task in which different implementations of the system are created from a reference implementation by replacing single components or clusters of components with alternative implementations available

in the CL. The process to replace a cluster or a component with an alternative implementation is graphically depicted in Figure 7.1-C.

7.3 System optimization methodology

The proposed optimization strategy is based on an extension of the extremal optimization theory [150].

7.3.1 Extremal optimization theory

The *Extremal Optimization* (EO) theory is a local search algorithm inspired by nature's self-organizing processes designed for combinatorial optimization problems. It explores a solution space trying to avoid sub-optimal solutions, thus driving the optimization towards real optimum.

Unlike genetic algorithms [151], which work with a population of candidate solutions, EO evolves a single solution and makes local modifications to the worst components. This is an important characteristic when considering the complexity of the Bayesian model presented in Chapter 6. The reliability model of a real system may easily include hundreds of nodes with their related CPTs. Working on big populations of such models can rapidly become computationally expensive. Another interesting feature is that the EO optimization process highly resembles the approach expert designers would use in manual optimization. Improvements to the system are searched by selectively removing critical components and replacing them with randomly selected alternatives. This is very different from other evolutionary techniques that look at combining "good" solutions in the attempt of improving the quality of the population. Moreover, EO is well suited for the optimization of multi-objective cost functions [152–154]. This is important in our case since the optimization process must take into account several design parameters such as reliability, area, power, performance, etc.

Finally, the EO is particularly effective in solving optimization problems, where near-optimum solutions are widely dispersed and separated by barriers in the search space [155]. This is a typical case in the particular optimization problem we are facing. Macro changes in the design (i.e., replacements of full clusters of components)

introduce high diversity in the generated systems that translate into a very sparse search space.

The main limitation of the basic formulation of the EO theory is that it is not intended to deal with a hierarchical organization of the components as the one introduced in Section 7.2. Therefore, the system optimization algorithm proposed in this chapter takes advantage of the basic principles of this theory but extends them to allow its application to complex hierarchical models.

7.3.2 Definitions and notation

This section introduces some basic definitions and notations required to describe the optimization strategy.

The design space $\Omega = \{S_0, S_1, \cdots, S_h\}$ *is the set of all possible system's implementations of the target design. Every implementation is a system description defined according to (7.1).*

S_0 represents the *reference implementation*, i.e., the initial design of the system that must be optimized (usually it does not include any fault tolerance mechanism). Starting from this implementation, the optimization strategy generates new implementations by selectively replacing worst components (nodes) or clusters of components based on the alternatives available in the CL.

The hierarchical organization of the CL allows us to introduce a hierarchical concept of distance between system implementations.

The k-level neighborhood *of a system implementation* $S \in \Omega$*, denoted as* $N_k(S)$*, is the set of implementations that can be created from S by replacing a single cluster of the system placed at the* k^{th} *hierarchical level of the CL.*

The k-level neighborhood hierarchically partitions the design space and therefore the optimization process. Informally, in a 2-level CL as the one reported in Figure 7.1-B we can identify two optimization levels. At a high level we have different implementations that differ for macro changes of the system due to replacements of clusters of components (e.g., changing the full microprocessor architecture, or

the full operating system architecture). At a fine grained level, selected nodes inside the clusters can be replaced based on the available alternatives to fine tuning the optimization. Managing hierarchical optimization is one of the main contributions of the optimization algorithm described in the next section. Every time a low level optimization reaches a local optimum, the optimization process must be able to go back to the higher hierarchical level and restart with a new macro change in the system.

Two different types of cost functions must be defined to implement the proposed optimization strategy.

The global cost function *$C(S)$ with $S \in \Omega$ is any generic function defined on any variable of S (i.e., N, E, Θ, P in (7.1)) that allows to compare two different implementations of the same system.*

This function is used to monitor the progress of the optimization process. In general the proposed algorithm supports any generic cost function defined over any variable in S. The simplest cost function that can be used when reliability is the only optimization goal is the AVF of the system. A description of the cost functions implemented for this work is reported in the next section together with a detailed description of the optimization algorithm.

The fitness *of a component or a cluster of components (cls) of a system S, denoted as $\lambda(cls,S)$, (with $cls \subset N$) is any generic function that permits individual components or clusters of components of the system to be assigned a quality measure with respect to their contribution to the global cost function.*

The fitness of a component is the criterion used during the optimization to select the components or the clusters of components to replace.

7.3.3 Optimization algorithm

Algorithm 2 describes the proposed optimization algorithm. The algorithm receives as an input the reference implementation of the system (S_0) and the available component library (CL) and returns an optimized implementation S_{best} and its related cost $C(S_{best})$.

Algorithm 2 System optimization algorithm.

Input: S_0, CL
Output: $S_{best}, C(S_{best})$

lev = k.
iter_no = 0
$S = S_0$
$S_{best} = S_0$
repeat
    λ_w=1;
    $cls_w = \emptyset$
    for each cluster cls at level lev **do**
        if $\lambda(cls) < \lambda_w$ **then**
            $\lambda_w = \lambda(cls, S)$
            cls_w = cls
        end if
    end for
    Generate $S' \in N_k(S)$ by selecting an alternative implementation of $cls \in CL$
    if $C(S') < C(S_{best})$ **then**
        $S_{best} = S'$;
    end if
    $lev = nextLevel(lev, S', S_{best})$
    $S = S'$
    iter1_no = iter1_no + 1;
until iter_no < MAX_ITER **and** stop(S_{best}) **not true**
return S^{best} and $C_k(S^{best})$

The optimization process is an iterative process (lines 5-21). At a high level of abstraction, at each iteration a new implementation of the system is generated by replacing one of the components (or a cluster of components) of the current system S with an alternative implementation from CL. The cost of this new implementation is evaluated to understand whether the introduced change leads toward a better implementation of the system or not. This iterative optimization process stops based on two different conditions (line 21):

1. the number of iterations (iter_no) reaches a maximum limit (MAX_ITER),
2. a *contract* on the identified implementation of the system is satisfied.

The contract is represented in Algorithm 2 by a generic *stop(·)* function. In our implementation we have defined a stop function that terminates the simulation when the estimated AVF of the system is lower than a user defined threshold (i.e., the system has reached the target reliability constraint), but other conditions can be easily defined. The first stop condition allows us to bound the duration of the optimization process, whereas the second allows us to define the goal of the optimization .

Lines 6-13 describe the process used by the algorithm to identify the component to replace at each iteration. To understand how this process works, it is important to recall that the system is organized into a k-level hierarchy of clusters (see Figure 7.1). At a given iteration the optimization process works at one of these k hierarchical levels. The hierarchical level sets the granularity of the replacement process. Let us consider the example of Figure 7.1. If the algorithm works at level 1, replacements of clusters take place at this level of the CL. If the algorithm works at level 2, components composing the level-1 clusters can be replaced. Algorithm 2 sets the initial level to k (line 1). This means that the algorithm initially privileges fine grained optimizations that do not introduce macro changes in the system and than moves toward more invasive optimizations. However, any level can be set to start the optimization process.

In lines 6-13 the fitness of every cluster belonging to the selected hierarchical level is evaluated for the current system implementation S in order to identify the cluster with worst fitness. A new implementation of the system S' is then generated by replacing the cluster with worst fitness with a random alternative implementation from CL (line 14). If the cost of this new implementation ($C(S')$) is lower than the

one of the best implementation ($C(S_{best})$), then the new version of the system is selected as local optimum.

At the end of each iteration, the algorithm evaluates whether the optimization has to continue at the current hierarchical level or it has to move up or down. This decision is taken at line 18 by the *nextLevel* function whose implementation is reported in Algorithm 3.

The function computes the next level by analyzing for each hierarchical level the number of iterations that have not produced improvements to the system using a set of counters (*count_worst[k]*). In case the new generated system implementation S' has introduced an improvement ($\delta \leq 0$ - line 1), the hierarchical level remains the same and the counter for the level is reset (line 13). Otherwise, the algorithm analyzes the counter of the layer (lines 3-11). If it is higher than a user defined threshold T (lines 4-5), it means we are not able to improve the system at the current level and therefore the algorithm moves up to a higher level. Otherwise (lines 7-10), the counter of the level is increased and then the algorithm tries to move down in the hierarchy to search if local changes can further improve the current solution. This overall idea behind this strategy is to privilege the lower levels of the hierarchy that correspond to small changes in the system moving up only when really required.

Algorithm 3 nextLevel function.

Input: k, S', S_{best}
Output: next k

```
1:  δ = C(S') − C(S_best)
2:  if δ > 0 then
3:      if count_worst[k] > T then
4:          count_worst[k]=0
5:          k = k - 1
6:      else
7:          count_worst[k]++
8:          if k < CL_LEVELS then
9:              k = k +1
10:         end if
11:     end if
12: else
13:     count_worst[k]=0
14: end if
15: return k
```

The current implementation of Algorithm 2 supports two different exploration strategies that translate into the definition of different cost and fitness functions described in the following subsections.

Optimization for best reliability

The system is optimized in order to maximize its reliability without taking into account other design parameters such as hardware area, software size, execution time, power consumption, etc. In this case the *cost function* is defined as the AVF of the system, i.e., the probability of a system failure given a hardware fault.

Resorting to the Bayesian model defined in Chapter 6, the AVF can be easily computed using Bayesian inference [156]. The hypothesis that a fault enters the system is emulated by setting a Bayesian evidence on the nodes of TD, and the posterior probability that at least one of the nodes of SD is in a failure state is computed using the Bayesian inference theory [156].

In a similar way, the *fitness* of a component can be computed by conditioning the Bayesian model with the hypothesis that the component is in a failure state and computing the posterior probability that the system fails given this event (i.e., at least one of the nodes of SD is in a failure state). When working with a cluster this probability is computed separately for each component of the cluster and the fitness is computed as the average probability over all components of the cluster.

Optimization for multiple objectives

The system is optimized in order to minimize four design dimensions:

- AVF
- hardware area,
- software size,
- execution time,
- power consumption

To compute the cost function, together with the AVF of the system that is computed using the same approach described in Section 7.3.3 the percentage increment of the remaining four parameters with respect to S_0 is computed. The five contributions are then combined with a simple weighted sum. The designer is free to assign the weights of each contribution depending on the optimization goals.

Similarly to the global cost function, the fitness function performs the same weighted sum but considering percentage increments of the nodes of the considered cluster.

7.4 Experimental Results

This section reports results obtained by the application of a C++ implementation of the proposed optimization algorithm to a set of realistic electronic systems.

7.4.1 Experimental setup

Experiments were performed by optimizing a set of microprocessor based systems running software applications on top of the Linux operating system. The optimization focused on the hardware and software architecture of the system, i.e., hardened technologies were not available to the designer. We considered two real microprocessor architectures:

1. ARM Cortex®-A15[1]: a high-performance ARMv7-A processor used in a variety of premium mobile and infrastructure applications.
2. Intel®-like i7-skylake[2]: a 64-bit microarchitecture that brings high performance and reduced power consumption.

Based on this hardware architecture we analyzed 10 different systems each running a different application software taken from the MiBench suite [92]. The following applications were selected: (1) Susan Smooth (susan_s), (2) Susan Edges (susan_e), (3) Susan Corners (susan_c), (4) Quick sort (QSort), (5) String search

[1] 32KByte L1 Instruction/Data caches, 1MByte L2 cache, 128 32-bit physical registers
[2] 32KByte L1 Instruction/Data caches, 1MByte L2 cache per core, 168 64-bit physical registers

(ssearch), (6) Secure Hash Algorithm (SHA), (7) JPEG decode (DJPEG), (8) JPEG encode (CJPEG), (9) AES decode (AES), (10) Fast Fourier Transformation (FFT).

Every system was modeled as described in Chapter 6 considering the occurrence of soft errors in 5 of the main memory arrays of the microprocessors (i.e., L2 cache, L1 Instruction/Data cache, Register File and Load/Store Queue). In order to build the model and to compute the CPTs of the different nodes we resorted to the tool-suite described in Chapter 6. The CL was organized in two hierarchical levels. The first level consists of the two microprocessor clusters, each comprising the 5 memory arrays considered in the study. At the second hierarchical level, we considered various state-of-the-art fault tolerance mechanisms that can be applied to hardware or software components. We selected protection mechanisms for which a clear estimation of the impact on the following design dimension is available in literature: (1) AVF, (2) hardware area, (3) software size, (4) software execution time, (5) power consumption.

Table 7.1 summarizes the fault-tolerance techniques that can be applied to hardware components. Overall, they improve tolerance to soft errors by: (i) modifying the circuit (LEAP, DICE and LEAP-DICE) [157], (ii) monitoring (SA-RDC) the data [158], or (iii) adding error correction codes (all ECC and Self-Immunity) [159]. A total of 8 techniques were selected. In a similar way, Table 7.2 reports the identified software implemented fault tolerance techniques. We selected 16 different techniques. Most of them (VARx techniques) are different combinations of variable duplication and cross checking validation techniques [160]. We also included a fault tolerance technique based on control and data flow assertions [161].

As reference implementation we considered systems based on the Intel®-like i7-skylake microprocessor not implementing any fault tolerance mechanisms. Both optimization strategies described in Section 7.3.3 were tested.

For each considered system the optimization algorithm was executed with a limit of 500 iterations without setting any stop condition based on the AVF of the system. This long simulation allowed us to clearly understand the dynamics of the optimization and to stress the limits of the optimization process. To deal with the randomness of the optimization process, every experiment was repeated 30 times and results were compared and averaged.

Table 7.1 Hardware fault tolerance techniques.

Technique	Description	Reliability Improvement (%)	Extra Time (%)	Extra Power Consumption (%)	Extra Hardware Area (%)
4 ECC	ECC applied to 4 registers over 32	+40%	+6%	+1%	+15%
16 ECC	ECC applied to 16 registers over 32	+91%	+8%	+2%	+22%
FULL ECC	ECC applied to all registers	+100%	+9%	+3%	+28%
Self-Immunity	Parity code for portion of registers	+91%	+4%	+0.20%	+11%
SA-RDC	Self-Adaptive caches using monitoring features	+99.98%	+7.40%	+43%	+0.39%
LEAP	Flip-Flop layout technique	30%	10%	10%	10%
DICE	Flip-Flop layout technique	90%	0%	60%	100%
LEAP-DICE	Flip-Flop layout technique	100%	2%	63%	100%

Table 7.2 Software implemented fault tolerance techniques.

Technique	Description	Reliability Improvement (%)	Extra Time (%)	Extra Power Consumption (%)	Extra Software Size (%)
VAR1	Variable Duplication and Cross-check mechanisms	+95%	+70%	+66%	+66%
VAR1+	Variable Duplication and Cross-check mechanisms	+95%	+66%	+64%	+64%
VAR1++	Variable Duplication and Cross-check mechanisms	+94%	+70%	+60%	+60%
VAR2	Variable Duplication and Cross-check mechanisms	+95%	+77%	+74%	+74%
VAR2+	Variable Duplication and Cross-check mechanisms	+95%	+73%	+70%	+70%
VAR2++	Variable Duplication and Cross-check mechanisms	+94%	+68%	+65%	+65%
VAR3	Variable Duplication and Cross-check mechanisms	+94%	+42%	+45%	+45%
VAR3+	Variable Duplication and Cross-check mechanisms	+94%	+37%	+41%	+41%
VAR3++	Variable Duplication and Cross-check mechanisms	+93%	+32%	+36%	+35%
VAR4	Variable Duplication and Cross-check mechanisms	+91%	+32%	+36%	+36%
VAR4+	Variable Duplication and Cross-check mechanisms	+91%	+27%	+33%	+33%
VAR4++	Variable Duplication and Cross-check mechanisms	+90%	+21%	+27%	+27%
VAR5	Variable Duplication and Cross-check mechanisms	+87%	+27%	+28%	+28%
VAR5+	Variable Duplication and Cross-check mechanisms	+84%	+24%	+25%	+25%
VAR5++	Variable Duplication and Cross-check mechanisms	+78%	+17%	+20%	+20%
SIFT	Fault Tolerance via control and data flow assertions	+95%	+40%	+20%	+20%

7.4.2 Results and discussion

Figure 7.2 shows how the optimization algorithm is able to optimize the selected systems for best reliability. The difference between the AVF of the reference implementation (blue bars) and the one of the optimized implementation (orange bars) is of several orders of magnitude, demonstrating the capability of the optimization algorithm to search the design space to find alternative implementations of the system with increased reliability. It is also interesting to report that, in 7 out of the 10 benchmarks (DJPEG, FFT, QSORT, SHA, SUSAN_E, SUSAN_C, SUSAN_S), the optimized reliability was achieved by moving from the Intel®-like i7-skylake microprocessor architecture to the ARM Cortex®-A15 architecture.

Figure 7.3 provides a deeper analysis of the optimization process reporting the absolute percentage improvement of the considered design parameters. This includes: AVF reduction, execution time overhead, power consumption overhead, hardware

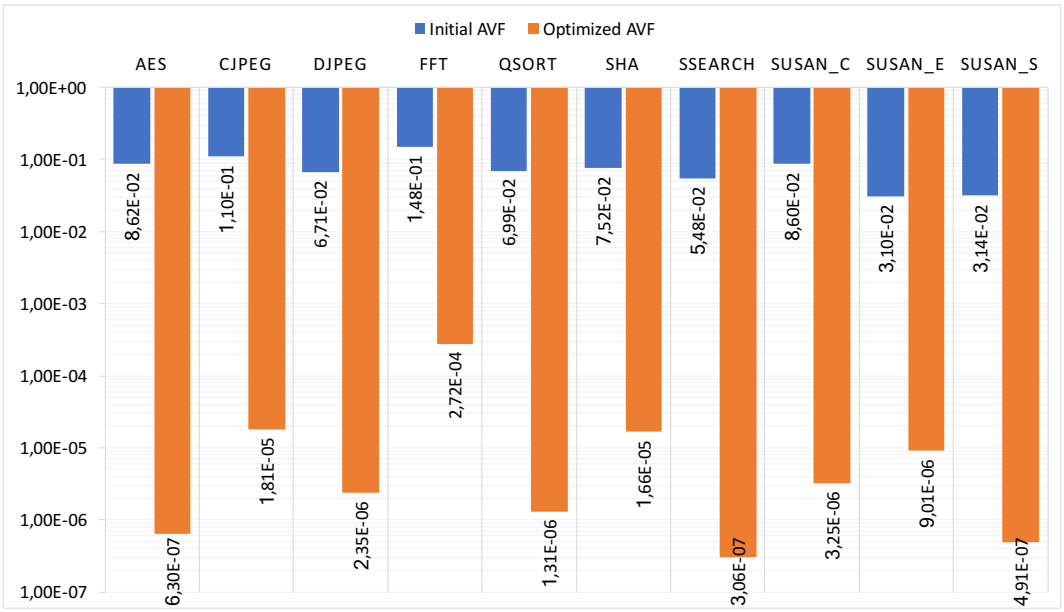


Fig. 7.2 AVF improvement (log scale) when optimizing for best reliability.

area overhead and software size overhead. The figure reports the average for each parameter over the 30 repetitions of each experiment. It is interesting to note that in all cases the reliability improvements have a significant impact on the other design dimension. This is a confirmation that, optimizing the system looking at a single objective is feasible but is probably not the best option. Figure 7.3 also gives an indication of how the optimization was achieved. The yellow and black bars indicate that in all cases the best reliability was achieved by a combination of hardware and software-implemented fault tolerance mechanisms.

Similar results can be computed changing the optimization strategy and optimizing the system for multiple objectives. As described in Section 7.3.3, this strategy tries to properly weight the reliability improvement with the overhead introduced by the use of different fault tolerance mechanisms. The multi-objective cost function used in this experiment (see Section 7.3.3) weighted the reliability as the 10% of the total value of the cost of an implementation with the remaining 90% equally shared between the other four design dimensions.

Figure 7.4 quantifies the benefit of the optimization process on the global AVF of the system. Improvements are still significant but obviously reduced with respect to the ones obtained when optimizing only for best reliability.

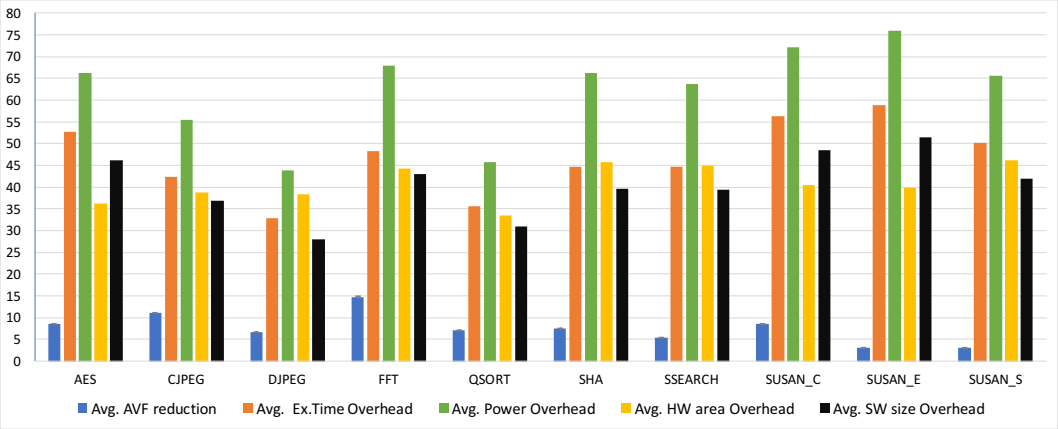


Fig. 7.3 Percentage improvement for all design dimensions when optimizing for best reliability.

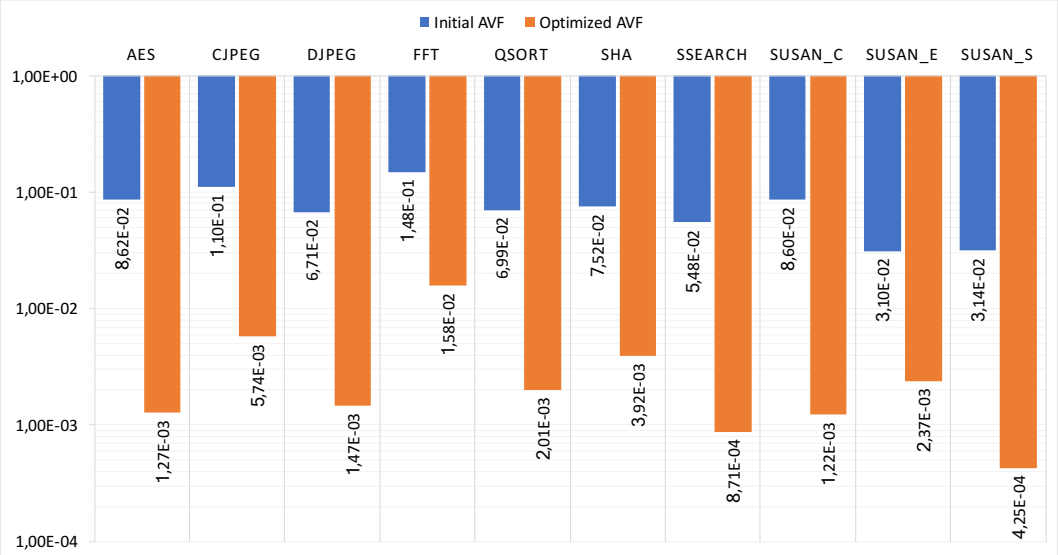


Fig. 7.4 AVF improvement (log scale) when optimizing for multiple objectives.

What is particularly interesting is to look at the effect of this optimization strategy on the different design parameters reported in Figure 7.5. The overhead of the execution time, power consumption, hardware area and software size is significantly reduced this time. In none of the cases it exceeds 15% with respect to the reference implementation with the exception of the hardware area overhead that in some cases increases up to 25%. This shows that, with a careful analysis, systems can reach high reliability levels without significantly penalizing the other aspects of the design. Moreover by playing with the weights of the cost function specific optimization goals (e.g., low area or low power) can be achieved.

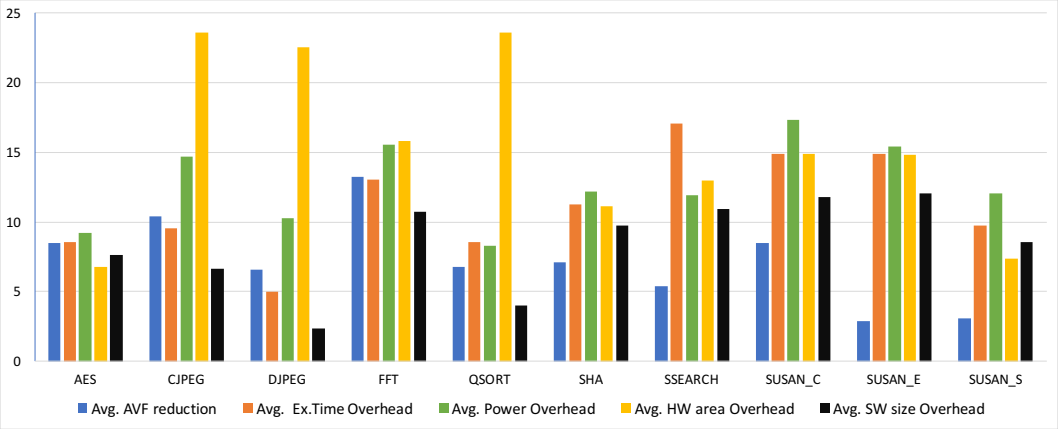


Fig. 7.5 Percentage improvement for all design dimensions when when optimizing for multiple objectives.

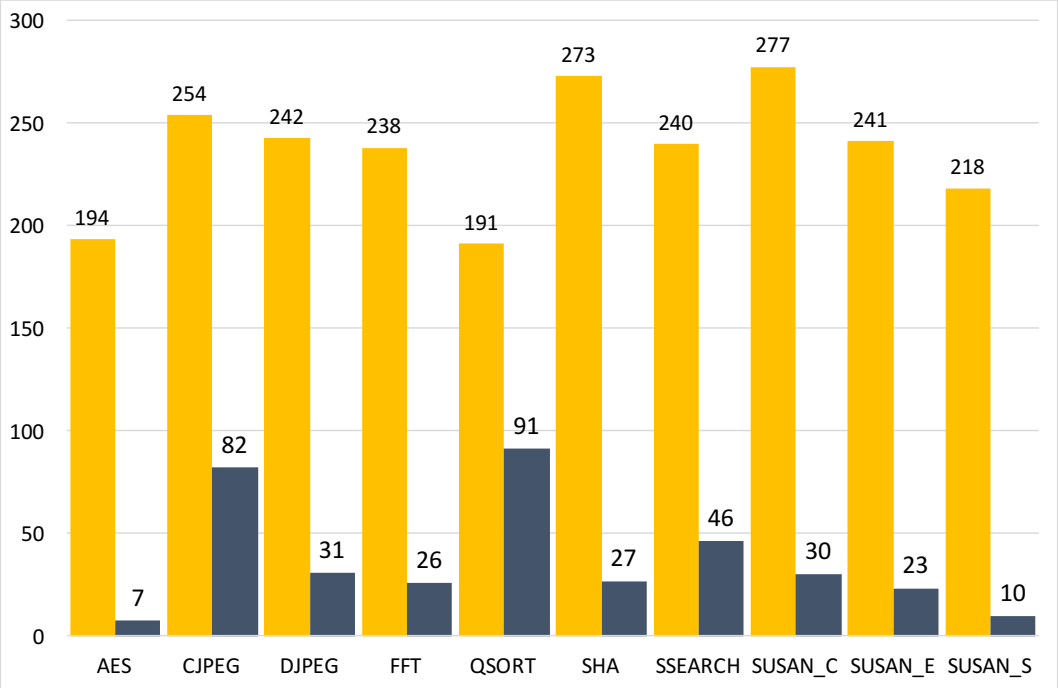


Fig. 7.6 Number of iterations before best solutions (500 iterations were simulated): optimization for best reliability (yellow bars) vs. optimization for multiple objectives (dark blue bars).

In order to provide a comprehensive analysis of the algorithm, we reported in Figure 7.6 the average number of iterations required by both optimization strategies to reach the optimum. The figure shows that, when optimizing for best reliability, a higher number of iterations is required before reaching the optimum compared to

the multi-objective optimization strategy. This can be ascribed to the fact that, when optimizing for multi-objectives, the multiple constraints introduced in the design reduce the degree of freedom in finding an optimal solution, while optimizing for a single parameters relaxes this constraint.

7.4.3 From benchmarks to real applications

The analysis conducted so far considers a realistic hardware architecture but is limited to simple benchmark software applications. To show the optimization algorithm at work on a real case we performed an additional experiment on a system running a real HPC application. We selected the Sierpinski framework[3], an open-source HPC application for the solution of hyperbolic partial differential equations used in several fluido dynamics simulators. The software application is very complex and its Bayesian model accounts for more than 800 nodes, thus representing a good candidate to stress the capability of the algorithm. The application is not designed for low-end microprocessors such as the ARM Cortex®-A15, therefore the optimization has was performed considering a single microprocessor architecture (Intel®-like i7-skylake).

Figure 7.7 and 7.8 show how the AVF of the system improves during the different iterations of the optimization process using the two considered optimization strategies. In this experiment, given the complexity of the system, the optimization has was limited to 250 iterations. The red line shows the trajectory of the best implementation while blue dots represent the AVF of all evaluated systems. Looking at the figure, it is interesting to report that, in this case, the reference implementation started from a significantly high AVF and thanks to the optimization process we were able to significantly increase the final reliability of the system. This is evident when looking at Figure 7.9 that shows the percentage increment for the 5 considered design parameters considering the two optimization strategies.

Interestingly, in this complex application, the system optimized for multiple-objectives is able to obtain very high reliability by working only on a few selected critical functions of the software layer suggesting that depending on the design the way the system can be optimized significantly changes. This further confirms

[3] https://www5.in.tum.de/sierpinski/

and motivates the benefit of an automatic optimization framework such as the one presented in this work.

Finally, when looking at the simulation time, we have to distinguish between the time required to build the components library and the one required to perform the actual optimization. For a complex application like the Sierpinski framework, the construction of the full library of components required a few days of simulation using the tool-suite presented in Chapter 6. This represents the most computational intensive task. Once the library is built, the optimization process required a few hours of simulation. All simulations were performed on a workstation equipped with a Intel i7 processor and 64MB of RAM.

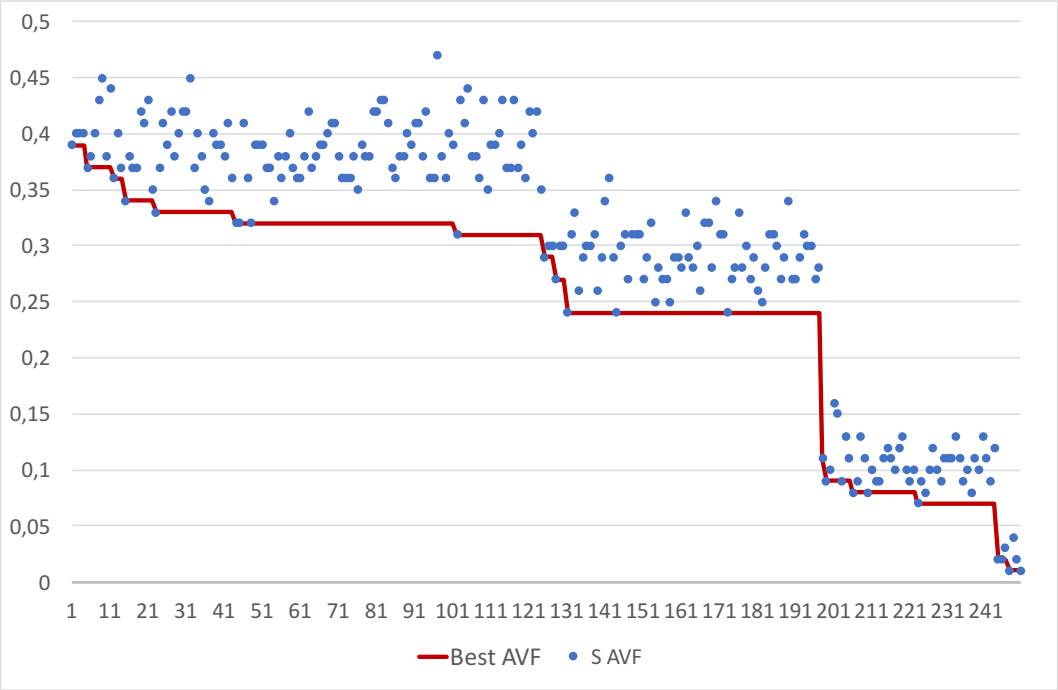


Fig. 7.7 AVF trajectory when optimizing the Sierpinski framework for best reliability.

7.5 Conclusion

This work presented a cross-layer multi-objective optimization algorithm for complex electronic systems based on a Bayesian reliability model of the system.

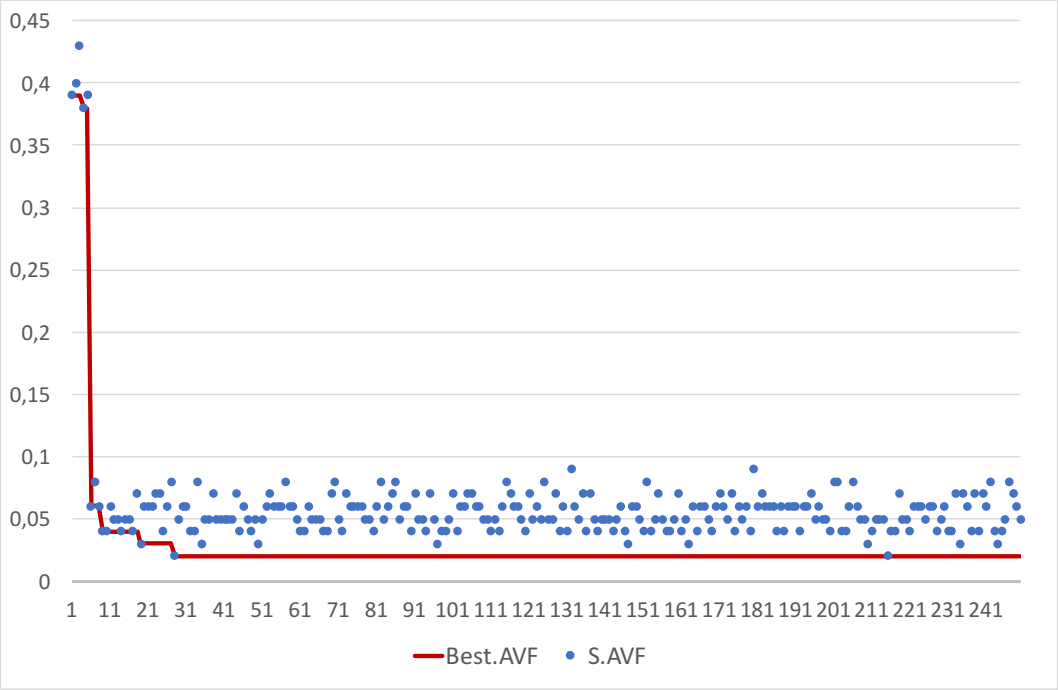


Fig. 7.8 AVF trajectory when optimizing the Sierpinski framework for multiple objectives.

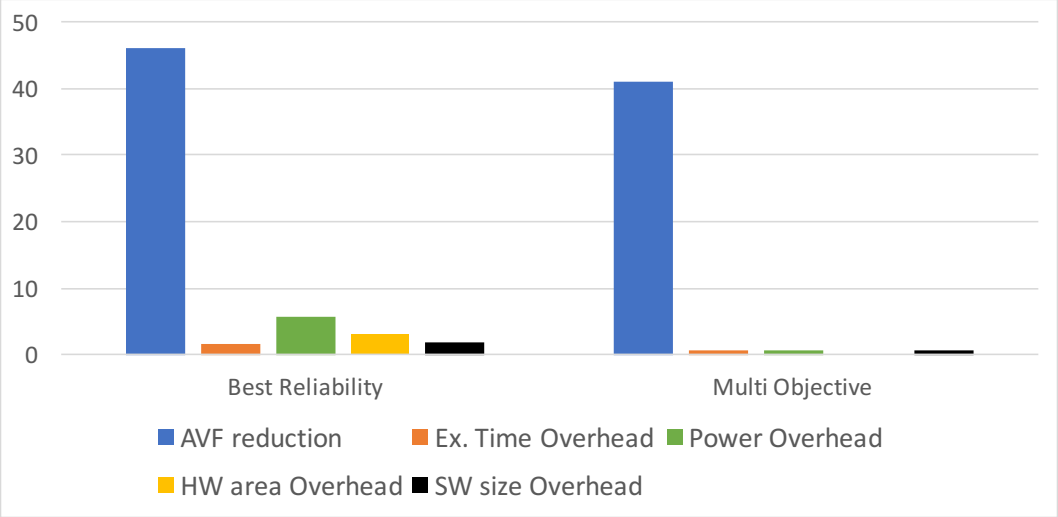


Fig. 7.9 Percentage improvement for all design dimensions when optimizing the Sierpinski framework using the two optimization strategies.

This work extends the extremal optimization technique to hierarchical organization of components. By applying the proposed algorithm in conjunction with a model for computing system reliability, it is possible to evaluate the effects that different HW/SW architectures and different protection mechanisms have on the

system design constraints such as reliability, execution time, power, hardware area and software size. The optimization can be executed based on a generic cost function in order to give the final user freedom to setup its specific optimization goals. Two optimization strategies, one for best reliability and one for multiple objectives were presented in this study.

The algorithm was tested on a set of systems based on realistic hardware running benchmark software and on a very complex system executing an open-source HPC application. In all cases the algorithm demonstrated its capability of optimizing the system.

Chapter 8

Conclusions

This Ph.D. thesis deeply analyzed reliability aspects related to digital systems. More specifically it contributes to the advance of the cross-layer reliability field introducing valuable methodologies addressing reliability estimation and reliability optimization during early design stages. Reliability is tackled for both single components of the system (i.e., FPGAs, GPUs, CPUs) and the system as a whole. In this thesis, the proposed methodologies were applied to use cases belonging to different computational domains: Autonomous Fault-Tolerant Systems built on top of FPGAs, General Purpose computing on Graphics Processing Units, Embedded Systems based on low-power microprocessors and HPC applications executed by high-end microprocessors.

The experimental results presented in this thesis have demonstrated that the proposed work constitutes a valuable alternatives to state-of-the-art solutions. In detail the proposed methodologies are characterized by high scalability, thus allowing to keep the pace with the increasing complexity of the systems. Moreover, the benefits introduced by the proposed methodologies hold the promise to face up the technological scaling, that, from the one hand, enhances computational power, while, on the other hand, contributes to unpredictable behaviors of the systems.

References

[1] Stefano Di Carlo, Giulio Gambardella, Paolo Prinetto, Daniele Rolfo, Pascal Trotta, and Alessandro Vallero. A novel methodology to increase fault tolerance in autonomous FPGA-based systems. In *IOLTS*, pages 87–92. IEEE, 2014.

[2] Alessandro Vallero, Sotiris Tselonis, Dimitris Gizopoulos, and Stefano Di Carlo. Microarchitecture Level Reliability Comparison of Modern GPU Designs: first findings. In *ISPASS*. IEEE, 2017.

[3] Alessandro Vallero, Alessandro Savino, Sotiris Tselonis, Nikos Foutris, Manolis Kaliorakis, Gianfranco Politano, Dimitris Gizopoulos, and Stefano Di Carlo. Bayesian network early reliability evaluation analysis for both permanent and transient faults. In *IOLTS*, pages 7–12. IEEE, 2015.

[4] Alessandro Vallero, Alessandro Savino, Gianfranco Politano, S. Di Carlo, Athanasios Chatzidimitriou, Sotiris Tselonis, Manolis Kaliorakis, Dimitris Gizopoulos, Marc Riera, Ramon Canal, A. Gonzalez, Maha Kooli, Alberto Bosio, and Giorgio Di Natale. Cross-layer system reliability assessment framework for hardware faults. In *ITC*, pages 1–10. IEEE, 2016.

[5] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[6] Rishad A. Shafik, Jimson Mathew, and Dhiraj K. Pradhan. *Introduction to Energy-Efficient Fault-Tolerant Systems*, chapter 1. Springer, 2014.

[7] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August. Design and evaluation of hybrid fault-detection systems. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 148–159. IEEE, 2005.

[8] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 264–275. IEEE, 2004.

[9] Cristian Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *2008 Annual Reliability and Maintainability Symposium*, pages 370–374. IEEE, 2008.

[10] R. A. Reed, M. A. Carts, P. W. Marshall, C. J. Marshall, O. Musseau, P. J. McNulty, D. R. Roth, S. Buchner, J. Melinger, and T. Corbiere. Heavy ion and proton-induced single event multiple upset. *IEEE Transactions on Nuclear Science*, 44(6):2224–2229, 1997.

[11] Charles Slayman. *JEDEC Standards on Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray Induced Soft Errors*, chapter 3. Springer, 2011.

[12] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *IEEE Transactions on Computers*, 49(3):230–245, Mar 2000.

[13] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: an architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, pages 243–247. IEEE, 2005.

[14] Shubu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[15] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 29–40. IEEE, 2003.

[16] Sridharan Vilas and David R. Kaeli. Using Hardware Vulnerability Factors to Enhance AVF Analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 461–472, New York, NY, USA, 2010. ACM.

[17] Qian Ding, Rong Luo, Hui Wang, Huazhong Yang, and Yuan Xie. Modeling the Impact of Process Variation on Critical Charge Distribution. In *2006 IEEE International SOC Conference*, pages 243–246. IEEE, 2006.

[18] P. Hazucha and C. Svensson. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *IEEE Transactions on Nuclear Science*, 47(6):2586–2594, 2000.

[19] A Survey of Techniques for Modeling and Improving Reliability of Computing Systems, 2015.

[20] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A Debardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta,

Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing Failures in Exascale Computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, May 2014.

[21] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[22] Y. C. Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307 vol.1. IEEE, 1996.

[23] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 214–224. IEEE, 2001.

[24] E. Rotenberg. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pages 84–91. IEEE, 1999.

[25] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 99–110. IEEE, 2002.

[26] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 25–36. IEEE, 2000.

[27] Seongwoo Kim and A. K. Somani. Area efficient architectures for information integrity in cache memories. In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pages 246–255. IEEE, 1999.

[28] Wei Zhang, S. Gurumurthi, M. Kandemir, and A. Sivasubramaniam. ICR: in-cache replication for enhancing data cache reliability. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pages 291–300. IEEE, 2003.

[29] W. Zhang. Replication cache: a small fully associative cache to improve data cache reliability. *IEEE Transactions on Computers*, 54(12):1547–1555, 2005.

[30] Makoto Sugihara, Tohru Ishihara, and Kazuaki Murakami. Task Scheduling for Reliable Cache Architectures of Multiprocessor Systems. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, 2007.

[31] G. Memik, M. T. Kandemir, and O. Ozturk. Increasing register file immunity to transient errors. In *Design, Automation and Test in Europe*, pages 586–591 Vol. 1. IEEE, 2005.

[32] Hamed Tabkhi and Gunar Schirner. Application-specific power-efficient approach for reducing register file vulnerability. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 574–577. IEEE, 2012.

[33] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Mixed-mode Multicore Reliability. *SIGARCH Comput. Archit. News*, 37(1):169–180, March 2009.

[34] Soontae Kim. Area-Efficient Error Protection for Caches. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 1, pages 1–6. IEEE, 2006.

[35] Kyoungwoo Lee, Aviral Shrivastava, Ilya Issenin, Nikil Dutt, and Nalini Venkatasubramanian. Mitigating Soft Error Failures for Multimedia Applications by Selective Data Protection. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 411–420, New York, NY, USA, 2006. ACM.

[36] Mahmut Yilmaz, Derek R. Hower, Sule Ozev, and Daniel J. Sorin. Self-Checking and Self-Diagnosing 32-bit Microprocessor Multiplier. In *2006 IEEE International Test Conference*, pages 1–10. IEEE, 2006.

[37] André DeHon, Nick Carter, and Heather Quinn. Final Report for CCC Cross-Layer Reliability Visioning Study. [Online] http://www.relxlayer.org/FinalReport?action=AttachFile&do=view&target=final_report.pdf, March 2011.

[38] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[39] Xilinx Corporation. *Automotive Driver Assistance Systems: Using the Processing Power of FPGAs - White paper (WP399)*, 2011.

[40] A. Hofmann, R. Wansch, R. Glein, and B. Kollmannthaler. An FPGA based on-board processor platform for space application. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pages 17–22, June 2012.

[41] Xilinx Corporation. *Partial Reconfiguration User Guide (UG702)*, 2013.

[42] X. Iturbe, K. Benkrid, T. Arslan, I. Martinez, M. Azkarate, and M.D. Santambrogio. A Roadmap for Autonomous Fault-Tolerant Systems. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 311–321, Oct 2010.

[43] Austin Lesea, Saar Drimer, Joseph J Fabula, Carl Carmichael, and Peter Alfke. The Rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs. *Device and Materials Reliability, IEEE Transactions on*, 5(3):317–328, 2005.

[44] Xilinx Corporation. *Continuing Experiments of Atmospheric Neutron Effects on Deep Submicron Integrated Circuits - White paper (WP286)*, 2011.

[45] Carl Carmichael, Earl Fuller, Phil Blain, and Michael Caffrey. SEU mitigation techniques for Virtex FPGAs in space applications. In *Proceeding of the Military and Aerospace Programmable Logic Devices International Conference (MAPLD)*, page C2, 1999.

[46] Cristiana Bolchini, Davide Quarta, and Marco D Santambrogio. SEU mitigation for SRAM-based FPGAs through dynamic partial reconfiguration. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, pages 55–60. ACM, 2007.

[47] Cristiana Bolchini, Antonio Miele, and Chiara Sandionigi. Autonomous Fault-Tolerant Systems onto SRAM-based FPGA Platforms. *Journal of Electronic Testing*, 29(6):779–793, 2013.

[48] Jie Zhang, Yong Guan, and Chunjing Mao. Optimal Partial Reconfiguration for Permanent Fault Recovery on SRAM-Based FPGAs in Space Mission. *Advances in Mechanical Engineering*, 2013, 2013.

[49] ITRS: International Technology Roadmap for Semiconductors. [Online:] http://www.itrs.net/Links/2011ITRS/Home2011.htm.

[50] Luca Cassano, Dario Cozzi, Sebastian Korf, Jens Hagemeyer, Mario Porrmann, and Luca Sterpone. On-line testing of permanent radiation effects in reconfigurable systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 717–720, March 2013.

[51] Michael E Imhof, Michael A Kochte, Eric Schneider, Hongyan Zhang, Jörg Henkel, and Hans-Joachim Wunderlich. Test strategies for reliable runtime reconfigurable architectures. *IEEE Transactions on Computers*, 62(8), 2013.

[52] Brendan Bridgford, Carl Carmichael, and Chen Wei Tseng. Single-event upset mitigation selection guide. *Xilinx Application Note XAPP987*, 2008.

[53] Zoltán Endre Rákossy, Masayuki Hiromoto, Hiroshi Tsutsui, Takashi Sato, Yukihiro Nakamura, and Hiroyuki Ochi. Hot-swapping architecture with back-biased testing for mitigation of permanent faults in functional unit array. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 535–540. IEEE, 2013.

[54] S-Y Yu and Edward J McCluskey. Permanent fault repair for FPGAs with limited redundant area. In *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*, pages 125–133. IEEE, 2001.

[55] Subhasish Mitra, W-J Huang, Nirmal R Saxena, S-Y Yu, and Edward J McCluskey. Reconfigurable architecture for autonomous self-repair. *IEEE Design & Test*, 21(3):228–240, 2004.

[56] John Lach, William H Mangione-Smith, and Miodrag Potkonjak. Low overhead fault-tolerant FPGA systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 6(2):212–221, 1998.

[57] Nikil Mehta and André DeHon. Variation and aging tolerance in FPGAs. In *Low-Power Variation-Tolerant Design in Nanometer Silicon*, pages 365–380. Springer, 2011.

[58] C. Bolchini, A. Miele, and C. Sandionigi. Increasing autonomous fault-tolerant FPGA-based systems' lifetime. In *Test Symposium (ETS), 2012 17th IEEE European*, pages 1–6, May 2012.

[59] Massimo Violante, Niccolo' Battezzati, and Luca Sterpone. *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer, 2011.

[60] Fernanda Gusmao Lima de Kastensmidt, Gustavo Neuberger, Renato Fernandes Hentschke, Luigi Carro, and Ricardo Reis. Designing fault-tolerant techniques for SRAM-based FPGAs. *IEEE Design & Test*, 21(6):552–562, 2004.

[61] Jonathan Heiner, Nathan Collins, and Michael Wirthlin. Fault tolerant ICAP controller for high-reliable internal scrubbing. In *Aerospace Conference, 2008 IEEE*, pages 1–10. IEEE, 2008.

[62] Ali Ebrahim, Khaled Benkrid, Xabier Iturbe, and Chuan Hong. A novel high-performance fault-tolerant ICAP controller. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pages 259–263. IEEE, 2012.

[63] Rino Micheloni, Alessia Marelli, and Roberto Ravasio. *Error correction codes for non-volatile memories*. Springer, 2008.

[64] Xilinx Corporation. *Virtex-4 FPGA User Guide - UG070*, 2008.

[65] S. Di Carlo, G. Gambardella, P. Prinetto, D. Rolfo, P. Trotta, and P. Lanza. FEMIP: A high performance FPGA-based features extractor amp; matcher for space applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–4, Sept 2013.

[66] Xilinx Corporation. *Virtex-4 FPGA Family Overview - DS112*, 2010.

[67] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: a simulation framework for CPU-GPU computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344. ACM, 2012.

[68] Alessandro Vallero, Dimitris Gizopoulos, and Stefano Di Carlo. SIFI: AMD Southern Islands GPU Microarchitectural Level Fault Injector. In *IOLTS*. IEEE, 2017.

[69] Paolo Rech, Laércio Lima Pilla, Philippe Olivier Alexandre Navaux, and Luigi Carro. Impact of GPUs parallelism management on safety-critical and HPC applications reliability. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 455–466. IEEE, 2014.

[70] Sotiris Tselonis and Dimitris Gizopoulos. GUFI: A framework for GPUs reliability assessment. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 90–100. IEEE, 2016.

[71] Athanasios Chatzidimitriou, Manolis Kaliorakis, Sotiris Tselonis, and Dimitris Gizopoulos. Performance-aware reliability assessment of heterogeneous chips. In *Proceedings of the IEEE VLSI Test Symposium*, 2017.

[72] N Farazmand, R Ubal, and D Kaeli. Statistical fault injection-based AVF analysis of a GPU architecture. In *IEEE workshop on silicon errors in logic*, 2012.

[73] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications. In *ISPASS*, pages 221–230. IEEE Computer Society, 2014.

[74] Jingweijia Tan, Yang Yi, Fangyang Shen, and Xin Fu. Modeling and characterizing GPGPU reliability in the presence of soft errors. *Parallel Computing*, 39(9):520–532, 2013.

[75] Hyeran Jeon, Mark Wilkening, Sridharan Vilas, Sudhanva Gurumurthi, and G Loh. Architectural vulnerability modeling and analysis of integrated graphics processors. In *Workshop on Silicon Errors in Logic-System Effects (SELSE), Stanford, CA*, 2012.

[76] Jingweijia Tan, Nilanjan Goswami, Tao Li, and Xin Fu. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *IISWC*, pages 226–235. IEEE Computer Society, 2011.

[77] Jeremy W. Sheaffer, David P. Luebke, and Kevin Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In Mark Segal and Timo Aila, editors, *Graphics Hardware*, pages 55–64. Eurographics Association, 2007.

[78] Ralph Nathan and Daniel J Sorin. Argus-g: A low-cost error detection scheme for gpgpus. In *Workshop on Resilient Architectures (WRA)*, 2010.

[79] Artem Durytskyy, Mohamed Zahran, and Ramesh Karri. Improving GPU Robustness by making use of faulty parts. In *ICCD*, pages 346–351. IEEE Computer Society, 2011.

[80] Hyeran Jeon and Murali Annavaram. Warped-dmr: Light-weight error detection for gpgpu. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 37–47. IEEE Computer Society, 2012.

[81] Jingweijia Tan, Zhi Li, and Xin Fu. Cost-effective soft-error protection for SRAM-based structures in GPGPUs. In Hubertus Franke, Alexander Heinecke, Krishna V. Palem, and Eli Upfal, editors, *Conf. Computing Frontiers*, pages 29:1–29:10. ACM, 2013.

[82] Ronak Shah, Minsu Choi, and Byunghyun Jang. Workload-dependent relative fault sensitivity and error contribution factor of GPU onchip memory structures. In *ICSAMOS*, pages 271–278. IEEE, 2013.

[83] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. Sassifi: Evaluating resilience of GPU applications. In *Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2015.

[84] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174. IEEE Computer Society, 2009.

[85] Nishant J. George, Carl R. Elks, Barry W. Johnson, and John Lach. Transient fault models and AVF estimation revisited. In *DSN*, pages 477–486. IEEE Computer Society, 2010.

[86] Nicholas J. Wang, Aqeel Mahesri, and Sanjay J. Patel. Examining ACE analysis reliability estimates using fault-injection. In Dean M. Tullsen and Brad Calder, editors, *ISCA*, pages 460–469. ACM, 2007.

[87] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[88] AMD Accelerated Parallel Processing OpenCL Optimization Guide available. [Online] http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide.pdf.

[89] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 502–506. European Design and Automation Association, 2009.

[90] Arijit Biswas, Paul Racunas, Razvan Cheveresan, Joel Emer, Shubhendu S. Mukherjee, and Ram Rangan. Computing Architectural Vulnerability Factors for Address-Based Structures. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 532–543, Washington, DC, USA, 2005. IEEE Computer Society.

[91] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[92] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, Dec 2001.

[93] Robert Baumann. Soft errors in advanced computer systems. *Design & Test of Computers, IEEE*, 22(3):258–266, 2005.

[94] Shekhar Borkar, Tanay Karnik, and Vivek De. Design and reliability challenges in nanometer technologies. In *Proceedings of the 41st annual Design Automation Conference*, pages 75–75. ACM, 2004.

[95] Dan Ernst, Shidhartha Das, Seokwoo Lee, David Blaauw, Todd Austin, Trevor Mudge, Nam Sung Kim, and Krisztián Flautner. Razor: circuit-level correction of timing errors for low-power operation. *IEEE Micro*, 24(6):10–20, 2004.

[96] Ramakrishna Vadlamani, Jia Zhao, Wayne Burleson, and Russell Tessier. Multicore soft error rate stabilization using adaptive dual modular redundancy. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 27–32. IEEE, 2010.

[97] Martin Dimitrov and Huiyang Zhou. Unified architectural support for soft-error protection or software bug detection. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 73–82. IEEE Computer Society, 2007.

[98] Nithin Nakka, Giacinto Paolo Saggese, Zbigniew Kalbarczyk, and Ravishankar K Iyer. An architectural framework for detecting process hangs/crashes. In *Dependable Computing-EDCC 5*, pages 103–121. Springer, 2005.

[99] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, and C. Tibaldi. PROMON: a profile monitor of software applications. In *8th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems 2005. DDECS 2005.*, pages 81–86. IEEE, 13-16 April 2005.

[100] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Towards understanding the effects of intermittent hardware faults on programs. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 101–106, June 2010.

[101] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the Propagation of Hard

Errors to Software and Implications for Resilient System Design. *SIGOPS Oper. Syst. Rev.*, 42(2):265–276, March 2008.

[102] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults. *SIGPLAN Not.*, 47(4):123–134, March 2012.

[103] Man-Lap Li, P. Ramachandran, U.R. Karpuzcu, S. K S Hari, and S.V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 105–116, Feb 2009.

[104] Sridharan V. and D.R. Kaeli. Eliminating microarchitectural dependency from Architectural Vulnerability. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 117–128, Feb 2009.

[105] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[106] C.T. Weaver, J. Emer, S.S. Mukherjee, and S.K. Reinhardt. Reducing the soft-error rate of a high-performance microprocessor. *Micro, IEEE*, 24(6):30–37, Nov 2004.

[107] A. Savino, S.D. Carlo, G. Politano, A. Benso, A. Bosio, and G. Di Natale. Statistical Reliability Estimation of Microprocessor-Based Systems. *Computers, IEEE Transactions on*, 61(11):1521–1534, Nov 2012.

[108] Helge Langseth and Luigi Portinale. Bayesian networks in reliability. *Reliability Engineering & System Safety*, 92(1):92–108, 2007.

[109] Levitin Yuan-Shun Dai and K.S. Trivedi. Performance and Reliability of Tree-Structured Grid Services Considering Data Dependence and Failure Correlation. *Computers, IEEE Transactions on*, 56(7):925–936, 2007.

[110] Sheng Zhai and Shu Zhong Lin. Bayesian networks application in multi-state system reliability analysis. *Applied Mechanics and Materials*, 347:2590–2595, 2013.

[111] Andrea Bobbio, Luigi Portinale, Michele Minichino, and Ester Ciancamerla. Improving the analysis of dependable systems by mapping fault trees into Bayesian networks. *Rel. Eng. & Sys. Safety*, 71(3):249–260, 2001.

[112] Nikos Foutris, Manolis Kaliorakis, Sotiris Tselonis, and Dimitris Gizopoulos. Versatile architecture-level fault injection framework for reliability evaluation: A first report. In *On-Line Testing Symposium (IOLTS), 2014 IEEE 20th International*, pages 140–145. IEEE, 2014.

[113] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSS: a full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*, pages 1050–1055. ACM, 2011.

[114] Marek J Druzdzel. SMILE: Structural Modeling, Inference, and Learning Engine and GeNIe: a development environment for graphical decision-theoretic models. In *AAAI/IAAI*, pages 902–903, 1999.

[115] Alessandro Vallero, Sotiris Tselonis, Nikos Foutris, Manolis Kaliorakis, Maha Kooli, Alessandro Savino, Gianfranco Politano, Alberto Bosio, Giorgio Di Natale, Dimitris Gizopoulos, et al. Cross-layer reliability evaluation, moving from the hardware architecture to the system level: A CLERECO EU project overview. *Microprocessors and Microsystems*, 39(8):1204–1214, 2015.

[116] J.Yoshida. Toyota Case: Single Bit Flip That killed. *EETimes*, October 2013.

[117] Todd M. Austin, Valeria Bertacco, Scott A. Mahlke, and Yu Cao. Reliable Systems on Unreliable Fabrics. *IEEE Design & Test of Computers*, 25(4):322–332, 2008.

[118] Subhasish Mitra, Kevin Brelsford, and Pia N Sanda. Cross-layer resilience challenges: Metrics and optimization. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1029–1034. IEEE, 2010.

[119] David W Coit, Tongdan Jin, and Naruemon Wattanapongsakorn. System optimization with component reliability estimation uncertainty: a multi-criteria approach. *IEEE transactions on reliability*, 53(3):369–380, 2004.

[120] Salvatore Distefano and Antonio Puliafito. Dependability Evaluation with Dynamic Reliability Block Diagrams and Dynamic Fault Trees. *IEEE Trans. Dependable Sec. Comput.*, 6(1):4–17, 2009.

[121] X. Li, S. V. Adve, Pradip Bose, and J. A. Rivers. SoftArch: an architecture-level tool for modeling and analyzing soft errors. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 496–505, June 2005.

[122] Niranjan Soundararajan, Angshuman Parashar, and Anand Sivasubramaniam. Mechanisms for bounding vulnerabilities of processor structures. In Dean M. Tullsen and Brad Calder, editors, *ISCA*, pages 506–515. ACM, 2007.

[123] Kristen R. Walcott, Greg Humphreys, and Sudhanva Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In Dean M. Tullsen and Brad Calder, editors, *ISCA*, pages 516–527. ACM, 2007.

[124] Lide Duan, Bin Li, and Lu Peng. Versatile prediction and fast estimation of Architectural Vulnerability Factor from processor performance metrics. In *HPCA*, pages 129–140. IEEE Computer Society, 2009.

[125] Arijit Biswas, Niranjan Soundararajan, Shubhendu S Mukherjee, and Sudhanva Gurumurthi. Quantized AVF: A means of capturing vulnerability variations over small windows of time. In *IEEE Workshop on Silicon Errors in Logic-System Effects*, 2009.

[126] Xin Fu, James Poe, Tao Li, and José A. B. Fortes. Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior. In *MASCOTS*, pages 147–155. IEEE Computer Society, 2006.

[127] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, and Paolo Prinetto. Static Analysis of SEU Effects on Software Applications. In *ITC*, pages 500–508. IEEE Computer Society, 2002.

[128] Sridharan Vilas and David R Kaeli. Using pvf traces to accelerate avf modeling. In *Proceedings of the IEEE workshop on silicon errors in logic-system effects*, pages 23–24, 2010.

[129] Philippe Weber, Gabriela Medina-Oliva, Christophe Simon, and Benoît Iung. Overview on Bayesian networks applications for dependability, risk analysis and maintenance areas. *Eng. Appl. of AI*, 25(4):671–682, 2012.

[130] David M Nicol, William H Sanders, and Kishor S Trivedi. Model-based evaluation: from dependability to security. *IEEE Transactions on dependable and secure computing*, 1(1):48–65, 2004.

[131] Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level. In Sven Overhage, Clemens A. Szyperski, Ralf H. Reussner, and Judith A. Stafford, editors, *QoSA*, volume 4880 of *Lecture Notes in Computer Science*, pages 108–126. Springer, 2007.

[132] Shunkun Yang, Minyan Lu, and Lin Ge. Bayesian Network Based Software Reliability Prediction by Dynamic Simulation. In *SERE*, pages 13–20. IEEE, 2013.

[133] Harshinder Singh, Vittorio Cortellessa, Bojan Cukic, Erdogan Gunel, and Vijayanand Bharadwaj. A Bayesian Approach to Reliability Prediction and Assessment of Component Based Systems. In *ISSRE*, pages 12–21. IEEE Computer Society, 2001.

[134] Hiroyuki Okamura, Michael Grottke, Tadashi Dohi, and Kishor S. Trivedi. Variational Bayesian Approach for Interval Estimation of NHPP-Based Software Reliability Models. In *DSN*, pages 698–707. IEEE Computer Society, 2007.

[135] Amrit L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on software engineering*, SE-11(12):1411–1423, 1985.

[136] Yu Jiang, Hehua Zhang, Xiaoyu Song, Xun Jiao, William N. N. Hung, Ming Gu, and Jiaguang Sun. Bayesian-Network-Based Reliability Analysis of PLC Systems. *IEEE Trans. Industrial Electronics*, 60(11):5325–5336, 2013.

[137] Wu Yueqin and Ren Zhanyong. Mission reliability analysis of multiple-phased systems based on Bayesian network. In *Prognostics and System Health Management Conference (PHM-2014 Hunan), 2014*, pages 504–508. IEEE, 2014.

[138] David Marquez, Martin Neil, and Norman E. Fenton. Improved reliability modeling using Bayesian networks and dynamic discretization. *Rel. Eng. & Sys. Safety*, 95(4):412–425, 2010.

[139] Antonio Filieri, Carlo Ghezzi, Vincenzo Grassi, and Raffaela Mirandola. Reliability analysis of component-based systems with multiple failure modes. In *International Symposium on Component-Based Software Engineering*, pages 1–20. Springer, 2010.

[140] Adam Zagorecki and Marek J Druzdzel. Knowledge Engineering for Bayesian Networks: How Common Are Noisy-MAX Distributions in Practice? In *ECAI*, page 482, 2006.

[141] David Heckerman, Dan Geiger, and David M Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3):197–243, 1995.

[142] Marc Riera, Ramon Canal, Jaume Abella, and Antonio González. A detailed methodology to compute Soft Error Rates in advanced technologies. In *DATE*, pages 217–222. IEEE, 2016.

[143] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, Nikos Foutris, and Dimitris Gizopoulos. Differential Fault Injection on Microarchitectural Simulators. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 172–182. IEEE, 2015.

[144] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[145] CLERECO. D4.2.2 – Software Characterization Methods, 2017.

[146] D.W. Coit, Tongdan Jin, and H. Tekiner. Review and comparison of system reliability optimization algorithms considering reliability estimation uncertainty. In *Reliability, Maintainability and Safety, 2009. ICRMS 2009. 8th International Conference on*, pages 49–53, July 2009.

[147] Bo Xing, Wen-Jing Gao, and T. Marwla. The applications of computational intelligence in system reliability optimization. In *Computational Intelligence for Engineering Solutions (CIES), 2013 IEEE Symposium on*, pages 7–14, April 2013.

[148] N. Wattanapongsakorn and S.P. Levitan. Reliability optimization models for embedded systems with multiple applications. *Reliability, IEEE Transactions on*, 53(3):406–416, Sept 2004.

[149] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel. Exploiting program-level masking and error propagation for constrained reliability optimization. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–9, May 2013.

[150] Stefan Boettcher and Allon Percus. Nature's way of optimizing. *Artificial Intelligence*, 119:275–286, 2000.

[151] David G Bounds. New optimization methods from physics and biology. *Nature*, 329:215–219, 1987.

[152] Min-Rong Chen and Yong-Zai Lu. A novel elitist multiobjective optimization algorithm: Multiobjective extremal optimization. *European Journal of Operational Research*, 188(3):637–651, 2008.

[153] Min-Rong Chen, Jian Weng, and Xia Li. Multiobjective extremal optimization for portfolio optimization problem. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 1, pages 552–556, Nov 2009.

[154] Ivanoe De Falco, Antonio Della Cioppa, Domenico Maisto, Umberto Scafuri, and Ernesto Tarantino. A multiobjective extremal optimization algorithm for efficient mapping in grids. In *Applications of Soft Computing*, pages 367–377. Springer, 2009.

[155] Jie Chen, Guo-Qiang Zeng, Kang-Di Lu, Wen-Wen Peng, Zheng-Jiang Zhang, and Yu-Xing Dai. Extremal optimization algorithm with adaptive constants dealing techniques for constrained optimization problems. In *Industrial Electronics and Applications (ICIEA), 2014 IEEE 9th Conference on*, pages 1745–1750, June 2014.

[156] George EP Box and George C Tiao. *Bayesian inference in statistical analysis*, volume 40. John Wiley & Sons, 2011.

[157] K Lilja, M Bounasser, S-J Wen, R Wong, J Holst, N Gaspard, S Jagannathan, D Loveless, and B Bhuva. Single-event performance and layout optimization of flip-flops in a 28-nm bulk technology. *IEEE Transactions on Nuclear Science*, 60(4):2782–2788, 2013.

[158] Shuai Wang, Jie Hu, and Sotirios G Ziavras. Self-adaptive data caches for soft-error reliability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(8):1503–1507, 2008.

[159] Hussam Amrouch and Joerg Henkel. Self-immunity technique to improve register file integrity against soft errors. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pages 189–194. IEEE, 2011.

[160] Eduardo Chielle, Fernanda Lima Kastensmidt, and Sergio Cuenca-Asensi. Tuning software-based fault-tolerance techniques for power optimization. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2014 24th International Workshop on*, pages 1–7. IEEE, 2014.

[161] Lei Xiong and Qingping Tan. A Configurable Approach to Tolerate Soft Errors via Partial Software Protection. In *Parallel and Distributed Processing with Applications Workshops (ISPAW), 2011 Ninth IEEE International Symposium on*, pages 260–265. IEEE, 2011.

[162] Saurabh Sinha, Greg Yeric, Vikas Chandra, Brian Cline, and Yu Cao. Exploring sub-20nm FinFET design with Predictive Technology Models. In *DAC Design Automation Conference 2012*, pages 283–288. IEEE, 2012.

[163] Neutron Flux Calculator (SEUTEST).

Appendix A

The SER of future devices

Riera et al. carried out a study on trend of future technologies SER [142]. This work was carried out by Universitat Politecnica de Catalunya in the context of the CLERECO project in which I was personally involved. Even if I did not contribute to this study, I think their work is very interesting and gives more relevance to the research topics presented in this thesis.

In [142] a comparison of SER is reported for different technologies, components and environmental conditions. More specifically, the technologies under study are: 22nm Bulk Planar, 22nm SOI Planar, 20nm Bulk FinFET, 16nm Bulk Planar and 14nm Bulk FinFET; the components are: SRAM cells implemented using 6 and 8 transistors (6T Cell and 8T Cell respectively), latches and logic gates NAND2 (NAND with 2 inputs) and NOT.

Figure A.1 shows results of SER for different technologies and components. SERs are in logarithmic scale and when looking at the bars of a component, such as the 6T cell, the higher SERs are for bulk planar and the lower ones are for bulk FinFET with SOI planar in the middle. Therefore, the most vulnerable technology is the bulk planar while bulk FinFET and SOI planar can reduce SERs up to 100x or even more in their lower technology nodes. This is due to the bigger sensitive area and the bigger collected charge of bulk planar.

Between components, both memory cells have similar results, the latch is a bit more reliable as it is vulnerable only 50% of the time (transparent mode) and the NAND2 has the lower SERs. Typical logic gates (NAND, NOR and NOT) usually have less sensitive nodes to strikes for each input combination, resulting in a total

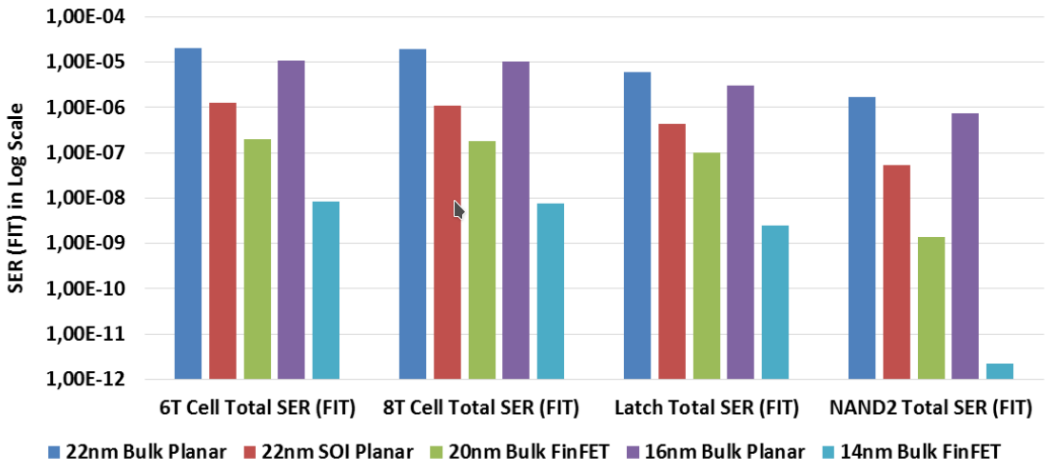


Fig. A.1 Technology comparison.

SER lower than other components. In addition, in bulk technology, lower technology

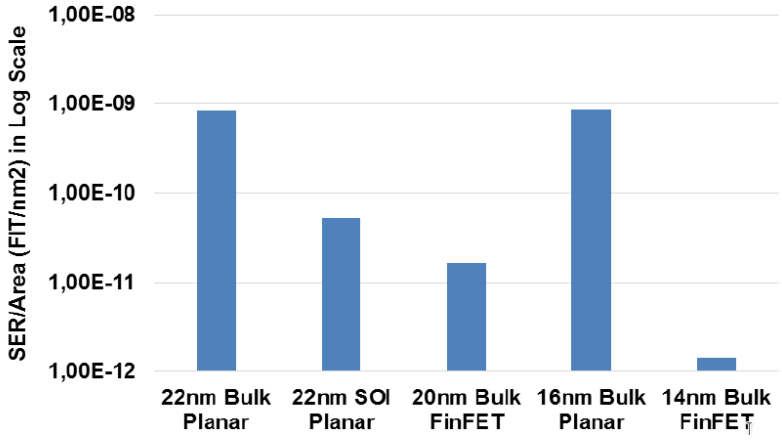


Fig. A.2 SERArea for a 6T SRAM cell.

nodes have lower SERs which may seem contradictory as in lower nodes *Qcrit* is usually reduced. However, the reduction in area has a stronger effect when the critical charge is already very low. Therefore, if we look at the SER/Area in Figure A.2, both nodes of bulk planar are quite similar, being slightly higher the node of 16nm. As an example, if we consider an SRAM chip with constant die area of 1.5 cm 2, the approximately SER of the 16nm chip would be 128694 FIT and 127549 FIT for the 22nm chip. In the case of FinFETs, our results show that the critical charge of the 14nm node is lower than the 20nm one. Therefore, adding the lower critical charge, the reduction in the sensitive area and the reduction in the collection efficiency, results in much lower SER values.

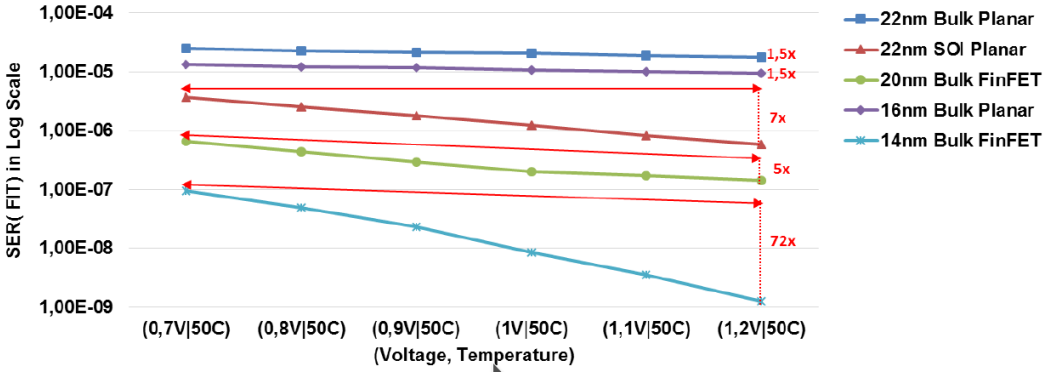


Fig. A.3 SER changing the voltage of a 6T SRAM cell.

Figure A.3 shows the SER of a 6T Cell varying technology and supply voltage. Results in logarithmic scale where lower values are better. SER increases with lower voltages since the critical charge becomes smaller. Therefore, it is easier to flip the value and the variation may be as high as 70x as can be seen with the red lines of the plot.

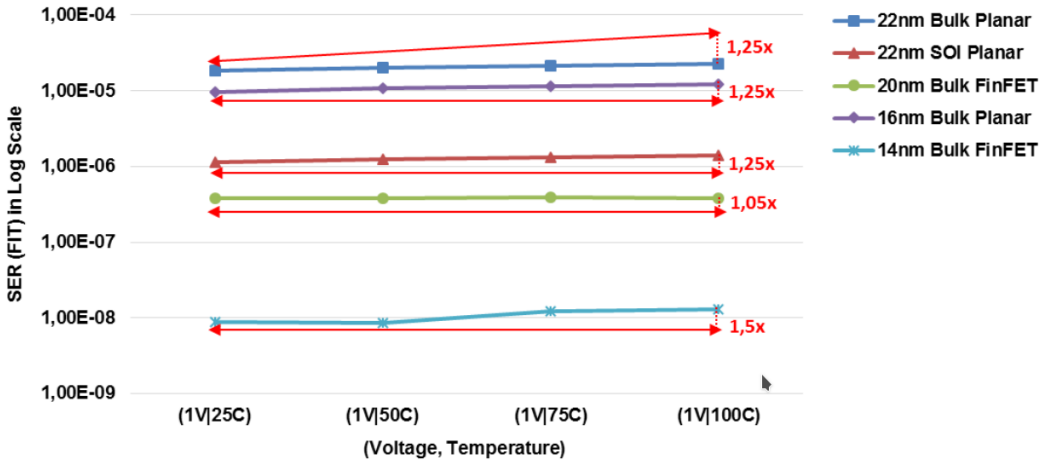


Fig. A.4 SER changing the temperature of a 6T SRAM cell.

SER increases with higher temperatures since the critical charge becomes smaller (Figure A.4). Even if seems that the variation is low it can be greater than 20% as can be seen with the red lines of the plot, but still has a low effect compared with the voltage variation. In the case of FinFET technology, the models used do not model the temperature accurately [162] so the variations are very low and slightly oscillating. Therefore, only the results from the nominal temperature (250C) should be used for FinFET technology.

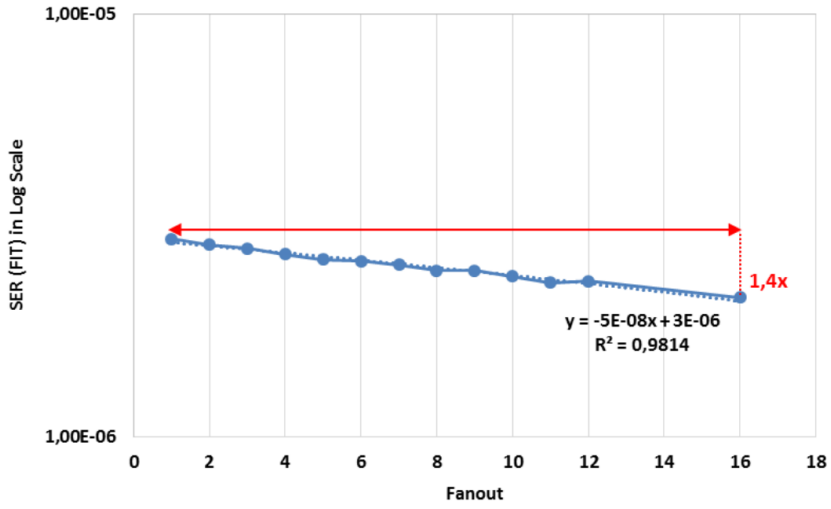


Fig. A.5 SER changing the fanout of a NOT logic gate.

Figure A.5 compares the soft error rates of the logic gate NOT build in 22nm bulk planar technology with different fanouts. SER is slightly reduced with higher fanout as there is more capacity in the output and the critical charge increases, with a variation that can be up to 1.5x. In this case, a linear model is suitable to fit of these values, with a squared R of 98%.

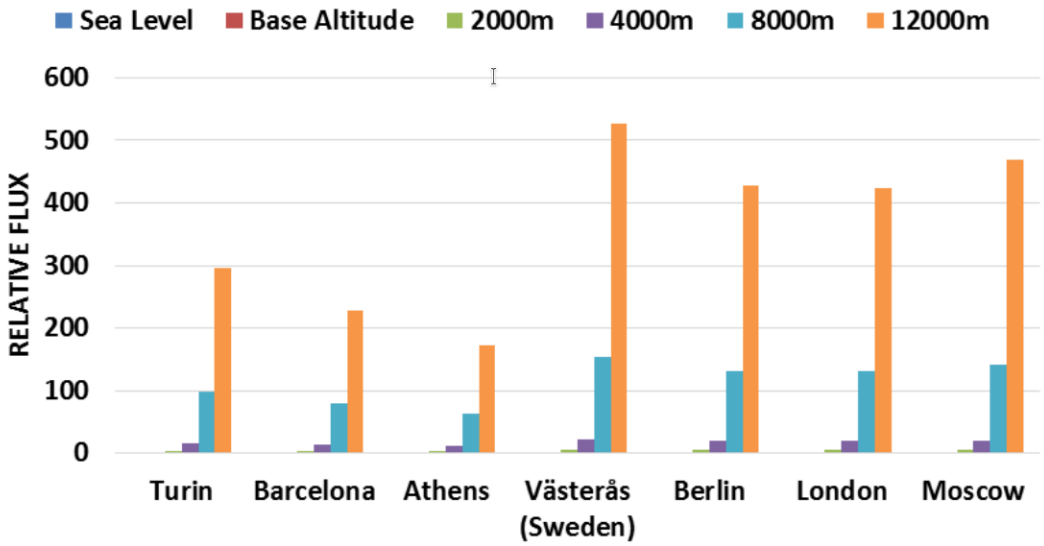


Fig. A.6 Relative neutron fluxes of different locations.

Finally, the location and the altitude also influence the SER since the $Flux$ changes (Figure A.6). The higher neutron fluxes are located in Västerås as is closer to the pole while the lower is in Athens which is nearer the equator. These relative

fluxes have been computed using the online calculator [163], which uses the JEDEC standard, with a medium solar activity (50%). The relative fluxes can be multiplied directly by the SER obtained with the reference flux to obtain the SER of the desired location. SER for a 6T cell at different locations and altitudes is shown in Figure A.7. The difference between cities is due the influence of the magnetic field of the earth, where cities near the equator have lower SERs. Moreover, there is an exponential increase of the SER when varying the altitude that can be as high as 650x.

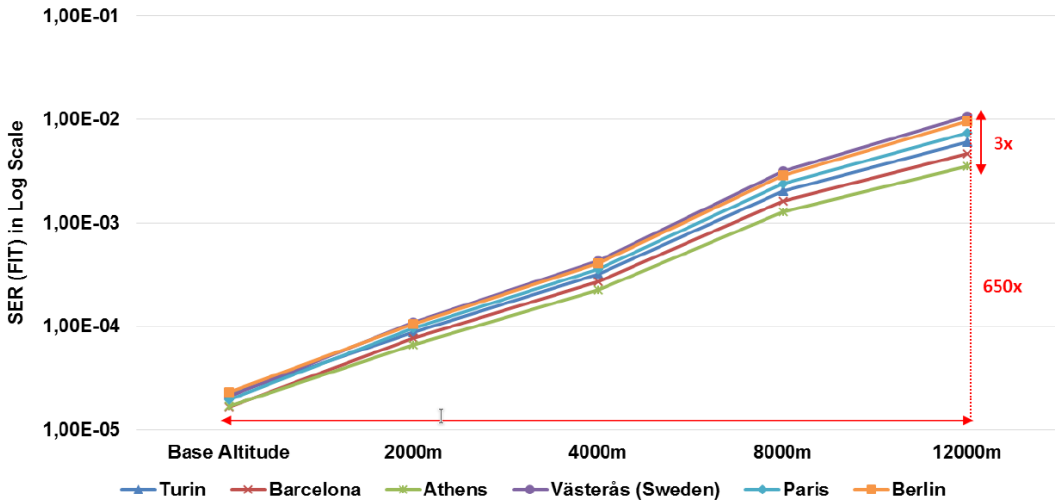


Fig. A.7 SERs depending on the Location and Altitude of a 6T SRAM Cell in 22nm Bulk Planar.